# Towards Availability and Maintainability Benchmarks: A Case Study of Software RAID Systems

Aaron Brown
Computer Science Division
University of California at Berkeley

1. REPORT DATE **JAN 2001**	2. REPORT TYPE	3. DATES COVERED **00-00-2001 to 00-00-2001**

4. TITLE AND SUBTITLE	5a. CONTRACT NUMBER
Towards Availability and Maintainability Benchmarks: A Case Study of Software RAID Systems	5b. GRANT NUMBER
	5c. PROGRAM ELEMENT NUMBER
6. AUTHOR(S)	5d. PROJECT NUMBER
	5e. TASK NUMBER
	5f. WORK UNIT NUMBER

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) **University of California at Berkeley,Department of Electrical Engineering and Computer Sciences,Berkeley,CA,94720**	8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)	10. SPONSOR/MONITOR'S ACRONYM(S)
	11. SPONSOR/MONITOR'S REPORT NUMBER(S)

12. DISTRIBUTION/AVAILABILITY STATEMENT **Approved for public release; distribution unlimited**

13. SUPPLEMENTARY NOTES

14. ABSTRACT

We introduce general methodologies for benchmarking the availability and maintainability of computer systems. Our methodologies are based on fault injection, used to purposefully compromise availability and to bring systems to a state where maintenance is required. Our availability benchmarks leverage existing performance benchmarks for workload generation and data collection, measure availability in terms of quality of service variation over time, and can produce results in both detail-rich graphical presentations or in distilled numerical summaries. Our maintainability benchmarks characterize several different axes of maintainability including the time, impact, and learning curve associated with maintenance tasks, and rely on the use of human experiments to capture the subtle interactions between system and administrator. We demonstrate and evaluate our methodologies by applying them to measure the availability and maintainability of the software RAID systems shipped with RedHat Linux 6.0, Solaris 7 for Intel Architectures, and Windows 2000 Server. We find that the availability benchmarks are powerful enough not only to quantify the impact of various failure conditions on the availability of these systems, but also to unearth their undocumented design philosophies with respect to transient errors and recovery policy. Similarly, the maintainability benchmarks draw clear distinctions between the systems on the time and learning curve metrics, and furthermore are able to identify key factors and design decisions influencing the maintainability of the three systems.

15. SUBJECT TERMS

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT **unclassified**	b. ABSTRACT **unclassified**	c. THIS PAGE **unclassified**	**Same as Report (SAR)**	**107**	

**Towards Availability and Maintainability Benchmarks:
A Case Study of Software RAID Systems**

by Aaron Brown

Research Project

Submitted to the Department of Electrical Engineering and Computer Sciences, University of California at Berkeley, in partial satisfaction of the requirements for the degree of **Master of Science, Plan II.**

Approval for the Report and Comprehensive Examination:

Committee

Professor David A. Patterson
Research Advisor

(Date)

* * * * * *

Professor Katherine Yelick
Second Reader

(Date)

Towards Availability and Maintainability Benchmarks:
A Case Study of Software RAID Systems[†]

Aaron Brown

University of California at Berkeley

`abrown@cs.berkeley.edu`

19 December 2000

Abstract

We introduce general methodologies for benchmarking the availability and maintainability of computer systems. Our methodologies are based on fault injection, used to purposefully compromise availability and to bring systems to a state where maintenance is required. Our availability benchmarks leverage existing performance benchmarks for workload generation and data collection, measure availability in terms of quality of service variation over time, and can produce results in both detail-rich graphical presentations or in distilled numerical summaries. Our maintainability benchmarks characterize several different axes of maintainability, including the time, impact, and learning curve associated with maintenance tasks, and rely on the use of human experiments to capture the subtle interactions between system and administrator.

We demonstrate and evaluate our methodologies by applying them to measure the availability and maintainability of the software RAID systems shipped with RedHat Linux 6.0, Solaris 7 for Intel Architectures, and Windows 2000 Server. We find that the availability benchmarks are powerful enough not only to quantify the impact of various failure conditions on the availability of these systems, but also to unearth their undocumented design philosophies with respect to transient errors and recovery policy. Similarly, the maintainability benchmarks draw clear distinctions between the systems on the time and learning curve metrics, and furthermore are able to identify key factors and design decisions influencing the maintainability of the three systems.

Table of Contents

List of Figures

Acknowledgements

x

Section 1

Introduction

There is a consensus emerging in parts of the systems community that the traditional focus on performance has become misdirected in today's world, a world in which the problems of availability, maintenance, and growth have become at least as important as peak performance, if not more so. One need only open up a recent issue of the *New York Times* or *Wall Street Journal* to see evidence of this fact—the number of stories focusing on recent outages of big e-commerce providers and the major business impact of those outages is staggering; furthermore, several of those outages have been reported as resulting from errors made by systems management staff [36]. Even from a financial standpoint, availability and manageability are important: the annual management costs for servers providing 24x7 service are typically reported as being several times that of the hardware itself [16] [19] [20].

The research community is beginning to recognize the importance of focusing on maintainability, availability, and growth as well. The attendees of the 7th HotOS workshop concluded that achieving "No-Futz Computing" (incorporating ideas of manageability, reliability, and availability, amongst others) is one of the most pressing challenges facing systems researchers today [39]. And, in his keynote at the 1999 FCRC conference, John Hennessy argued the same point, insisting that "performance should be less of an emphasis. Instead, other qualities will become crucial: *availability* [...], *maintainability*, [... and] *scalability*. [...] For servers—if access to services on servers is the killer app—availabil-

ity is *the* key metric" [23]. Furthermore, the traditional "scalability problem," of creating and efficiently using large massively-parallel systems, is giving way to what we call the "evolutionary growth problem": constructing large-scale servers that can be incrementally expanded using newer, heterogeneous components.

Despite this initial interest, it will take a significant shift in focus for both academic and industrial researchers to shrug off the traditional sole consideration of performance and to dedicate themselves to understanding and addressing the challenges of availability, maintainability, and evolutionary growth. But how are we to achieve this shift? History suggests a possible answer: *benchmarks*. Throughout the modern history of computer science, benchmarks have time and again acted as a revolutionary force. Across the field, benchmarks have introduced the notion of scientific comparison and have driven the evolution of system architecture and design. More importantly, they have acted as catalysts for new research areas by providing the fundamental metrics and measurement tools needed to evaluate advances in those areas.

Consider several motivating examples. In the late 1980's, the field of computer architecture was revolutionized by the development of a quantitative approach to CPU design [24] in conjunction with the roughly concurrent introduction of the SPEC CPU benchmark suite [46]. These new benchmarks and metrics provided (for the first time) a scientific method for comparing CPU designs, and were a direct influence on the development and popularization of the now-ubiquitous RISC architectures. Turning to another field, the first database (DBMS) benchmarks released by the Transaction Processing Performance Council (TPC) took a field that was fast trading scientific credibility for fabricated marketing glitz and thrust it back onto track [3] [43]. According to industry insiders, the TPC benchmarks have played an even more important role by driving the evolution of DBMS architecture and design, thereby providing increased performance and functionality to customers [22]. Today, the TPC benchmarks provide the canonical yardstick by which nearly all database systems are measured and compared.

We believe that the time has come for benchmarks to once again step up as a shaping force in the field of computer systems research and development. It is time for benchmarks to expand past the space of performance measurement and into the realm of quantifying availability, manageability, and growth. Once this transition occurs, we believe that

research into these areas will become significantly more tractable, and that research progress will naturally follow. As part of the Berkeley ISTORE project [9], we have taken on the challenge of effecting this transition by building reproducible, cross-platform "AME" benchmarks for Availability, Maintainability, and Evolutionary Growth, the three challenge areas laid out by Hennessy [23]. This report presents our first steps toward that goal.

We have chosen to focus initially on availability and maintainability, as we suspect that the task of measuring evolutionary growth boils down to measuring availability and maintainability as system scale is increased. We have developed metrics and general methodologies for quantifying availability and maintainability, which we will introduce in later sections of this report. In order to validate and illustrate our approaches, we have applied our methodologies to measure the availability and maintainability of the software RAID-5 implementations that ship with three popular PC server operating systems: RedHat Linux 6.0, Solaris 7 Server for Intel Architectures, and Windows 2000 Server. This case study of RAID-5 is obviously just an example, and we hope our approach will inspire others to benchmark the availability and maintainability of other subsystems.

We chose software RAID as a case study for several reasons. First, software RAID implementations are included with many commercial OS releases (such as the server editions of Solaris 7 and Windows 2000) and with all of the major free UNIX-like operating systems, including Linux, which is being increasingly deployed for Internet service applications. More importantly, RAID has well-defined availability goals and a well-understood set of maintenance tasks, making it an ideal candidate application for our benchmarks. Also, it is not unusual to find software RAID underlying many Internet service applications that demand 24x7 availability, and thus the availability and maintainability of the RAID implementation play an important role in that of the service application itself. Finally, although there is agreement on general features of a RAID-5 system, availability/maintainability benchmarking can highlight RAID implementation decisions that are important to applications but that are not measured or even mentioned today. For example, our tests revealed important differences in how a RAID system distinguishes between a disk failure and a temporary glitch, and in how a RAID system handles routine maintenance like disk replacement.

Our benchmarking case study revealed several interesting results. In studying the availability of the software RAID systems, we found significant differences in implementation philosophy between the various OS implementations. The major differences in philosophy between the systems can be classified along two axes: the first measures the system's paranoia with respect to transient errors, while the second measures the relative priorities placed on preserving application performance versus quickly rebuilding redundancy after a failure. On these axes, the Linux software RAID implementation is paranoid about transients but values application I/O performance more than fast post-failure reconstruction. Solaris falls at the opposite end of both spectrums, demonstrating a near-complete tolerance for transient errors and emphasizing fast reconstruction despite its potential impact on application performance. Windows 2000 falls between Linux and Solaris, although it lies closest to the Solaris end of the spectrum: it tolerates a set of transient errors that is only slightly less robust than Solaris's, and demonstrates a reconstruction philosophy that is similarly aggressive but more workload-aware than Solaris's. The fact that our availability benchmarks could reveal these philosophies despite treating the implementations as black boxes highlights the power of the methodology.

Turning to the maintainability benchmarks, we again found interesting results. Despite being based on a very rudimentary implementation of our methodology, our case study was able to uncover and evaluate the differences in the maintainability approaches taken by Solaris, Linux, and Windows 2000. In particular, it identified several key qualitative factors in software-RAID-system maintainability, including the system's approach to naming disks and the design and scriptability of its administrative interfaces. The case study also generated quantitative metrics comparing the overall maintainability of the three systems on the particular task of detecting and repairing a failed disk. Although the number of subjects in our experiments was too small to draw solid statistically-supported conclusions, our data suggests that on this task, using human interaction time as a metric, Solaris was the most time-efficient system to maintain, followed by Linux and then Windows. Using a different metric that we call "learning curve", which is a metric that relates to the difficulty and complexity of maintaining the system, the ordering suggested by our data changes, with Windows taking first place, followed closely by Solaris and trailed by Linux. These varying results reinforce an important principle behind our proposed meth-

odology: that the benchmarking technique be capable of measuring maintainability along several axes, and that it must be the end user of the benchmark results that chooses which axes are most important. Finally, although our results in this benchmark are not as precise as might be desired, they represent an initial, successful step in the development of maintainability benchmarks, and the fact that they could be achieved at all again shows the promise of our methodology.

The remainder of this report is organized as follows. Section 2 considers availability benchmarks, introducing our methodology, presenting the availability results of the software RAID case study, and discussing their implications. Section 3 does the same for maintainability benchmarks. We discuss related work in Section 4, and wrap up with our overall conclusions in Section 5.

Section 2

Benchmarks for Availability

In this section, we develop a general methodology for benchmarking system availability, and illustrate its effectiveness via a case study measuring the availability of software RAID systems on Linux, Solaris, and Windows 2000.

2.1 A General Methodology for Availability Benchmarking

We begin by defining availability and the metrics that can be used to report it, then consider how to construct benchmarks to produce the desired metrics, and finally describe how the results of those benchmarks can be reported and analyzed.

2.1.1 Availability: definitions and metrics

The term *availability* carries with it many possible connotations. Traditionally, availability has been defined as a binary metric that describes whether a system is *up* or *down* at a single point of time. A traditional extension of this definition is to compute the percentage of time, on average, that a system is available (*up*)—this is how availability is defined when a system is described as having 99.999% availability, for example.

We take a different perspective on availability, motivated largely by the fact that modern systems already do quite well on the traditional availability metric, indicating that this metric is not sensitive enough to reveal the intricacies of a system's availability behavior. Our perspective begins by first viewing availability as a spectrum, and not a binary metric. Systems can exist in a large number of degraded, but operational, states between *down* and

up. In fact, systems running in degraded states are probably more common than "perfect" systems [5], especially in the fast-growing world of online service provision where economic pressures encourage deployment of less-well-tested commodity SMP- and cluster-based servers rather than expensive fault-tolerant machines. An availability metric must therefore capture these degraded states, measuring not only whether a system is up or down, but also its efficacy, or the quality of service that it is providing.

Second, availability must not be defined at a single point in time or as a simple average over all time. It must instead be examined as a function of the system's quality of service over time. To motivate this, consider that from a user's perspective, there is a big difference between a system that refuses requests for two seconds out of every minute and one that is down for one whole day every month, even though the two systems have approximately the same average uptime. Any benchmark of availability must be able to capture the difference between those two systems.

Combining these two requirements, we propose that availability be measured by examining the variations in system quality of service metrics over time. The particular choice of quality of service metrics depends on the type of system being studied. Two obvious metrics that apply to most server systems are performance and degree of fault-tolerance. For a web server, these metrics would map to requests satisfied per second (or perhaps latency of request service) and the number of failures that can be tolerated by the storage subsystem, network connection topology, and so forth. Other possible metrics might include:

- *completeness*: consider a system like the Inktomi search engine that tolerates failures by returning search results that cover only the remaining available parts of its database [18];
- *accuracy*: a system that must perform a large computation in a fixed amount of time (e.g., decoding real-time media) might sacrifice accuracy in the computation when running in degraded mode; and
- *capacity*: to maintain other metrics while in a degraded state, a system might limit the number of clients or jobs it will accept, or might discontinue less-essential services.

We discuss how these time-dependent quality-of-service measurements might be concretely represented as graphs and numerical summary statistics in Section 2.1.3, below.

2.1.2 Towards an availability benchmarking methodology

Having selected the availability (quality-of-service) metrics for a given type of system, our next challenge is to accurately and reproducibly measure them in a controlled benchmarking environment. Doing this is complicated, because typical benchmark environments are explicitly designed to prevent the kinds of exceptional behavior that would cause availability to be affected in real-world systems.

Thus, in order to perform availability benchmarks, it is necessary to have a benchmark environment that provides a means of generating fault-provoking stimuli and *maintenance events* and applying them to the system under test. (A maintenance event is any action taken by a human administrator to maintain, repair, or upgrade the system.) The primary technique that enables such an environment to be constructed is direct *fault injection* into the system under test [4] [11]. For example, disk failures in a storage array can be simulated, memory can be artificially corrupted, processes can be killed, power glitches can be simulated, network links can be broken, and so forth. Fault injection need not be limited to hardware faults, however: stimuli such as load spikes, invalid client/user requests, and other workload-driven ways of triggering boundary conditions are also reasonable events to simulate.

To build an availability benchmark, we also need a way to generate a realistic workload and to measure the appropriate quality of service metrics. Our task can be simplified by leveraging the extensive efforts at fair workloads from the performance benchmarking community. When they exist, we simply use existing performance benchmarks to generate a representative workload for the type of system under test, and to measure the desired metrics at a single point in time. If the granularity of measurement is too coarse, these workload-generating performance benchmarks may need to be adapted to run continuously, repeatedly measuring the desired metric over small time periods. The system under test may also need to be modified to measure certain metrics (such as accuracy or completeness) that are typically neglected by performance benchmarks. Finally, if workload-

induced faults such as load spikes or erroneous inputs are to be used, the workload generator may need further modification to inject these faults at the appropriate time.

Given a benchmark environment supporting fault injection and a performance benchmark configured as both a continuous workload generator and a quality of service data collector, running an availability benchmark consists of two steps. First, the workload generator is run without injecting faults and several traces of the values of the desired metrics are recorded. This step establishes a baseline measurement for a non-faulty system. Second, the workload generator is run while simultaneously injecting a *fault workload*, and again a trace of the values of the desired metrics is recorded. This second step is key, since it produces a trace of the behavior of the system's quality of service over time in response to various faults, which is exactly the time-dependent availability metric that is desired.

The only part of the methodology we have not yet discussed is the content of a fault workload. As its name suggests, a fault workload is a collection of faults and maintenance events designed to mimic a real-world failure situation.

We see the need for two different kinds of fault workloads, described in the following two sections, roughly corresponding to traditional micro- and macro-benchmarks:

Single-fault workloads. The first kind of fault workload is the availability analogue of a performance microbenchmark. A single-fault workload, as its name implies, consists of just a single fault: once the system under test has reached steady-state, a single fault is injected—such as a disk sector write error—and the system's behavior (as reflected in the quality of service metrics) is recorded. Intervention of a human administrator in response to the fault is not allowed. Like performance microbenchmarks, single-fault availability benchmarks are most useful for studying isolated pieces of a system and for uncovering design decisions, design flaws, and bugs. Their scope is broader than performance microbenchmarks, however, since a single fault can often have a ripple effect and affect a system as much as a multi-fault workload.

Multi-fault workloads. The second kind of fault workload is the availability equivalent of a performance macrobenchmark. Multi-fault workloads consist of a series of faults and maintenance events designed to mimic real-world fault scenarios, for example, a disk fail-

ure in a RAID system followed by replacement of the failed disk followed by a write failure while reconstructing the array. Like traditional application performance macrobenchmarks, multi-fault workloads are useful for building availability benchmarks designed to help select or evaluate new systems, and to identify potential weaknesses in existing systems that need to be addressed. They are also very useful for studying the behavior of the system under pathological failure conditions, as in the RAID example above.

A challenging problem in developing benchmarks based on multi-fault workloads lies in how to realistically and reproducibly simulate the behavior of a human administrator in maintaining the system and in responding to failures originating from fault injection. Such *maintenance events* cannot be ignored, as very few modern systems are truly self-maintaining and most will require human intervention to complete the scenarios. The simplest solution is to run the benchmark with simulated maintenance events that represent correct and appropriate human maintenance actions. The approach is motivated by our focus here on availability: the approach captures the delays that would occur even in the best case, but does not attempt to quantify the effects of human variability or error-proneness, factors that are best left for a maintainability benchmark such as those we describe in Section 3. The simulated maintenance events could be specified *a priori* by the system designer, or extracted from logs of typical administrator activity. For example, one might characterize the statistical distribution of best-case human response times between a reported disk failure and the replacement of that disk. Such a model can then be used to direct the simulated human intervention during the benchmark run, although doing this in a truly cross-platform manner is an enormous challenge. There is a parallel here to performance benchmarks designed for systems that require human interaction; often in these benchmarks, a script plays back what a person would type in response to prompts. As with scripted performance benchmarks, we must be careful that our simulated maintenance events do not over-constrain the system, for example by playing back events at an unrealistically high rate.

Note that disk improvements over the years mean that disks no longer fail fast: the classic head crash of operating systems lore almost never happens today, as disks have become physically smaller and their mean time between failures has increased from
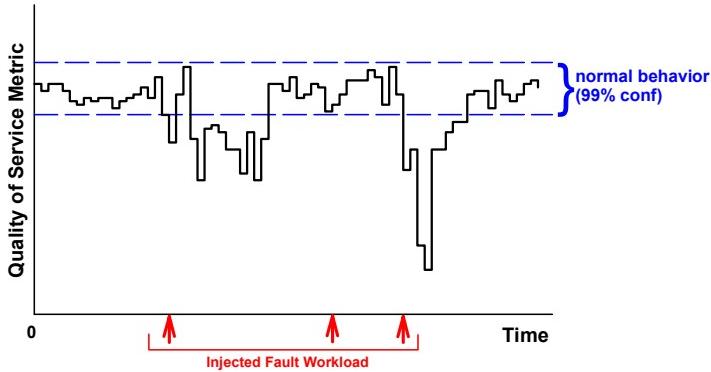
Figure 1: Example availability graph. The graph shows an example of the variation in an application quality of service metric (on the vertical axis) over time (on the horizontal axis), as faults are injected into the system (the faults are represented by heavy arrows). The dashed lines define a 99% confidence interval around the system's normal (non-faulty) behavior.

50,000 to 1,000,000 hours. Observations of the Tertiary Disk (TD) system at UC Berkeley, a large disk and web server farm, suggest that modern components start acting erratically rather than failing fast, and so a system administrator is much more likely to "fire" and replace an erratic component than to wait for it to fail completely [47]. We feel it is important to capture this type of activity in any synthetic model of administrator behavior.

2.1.3 Analyzing and reporting availability benchmark results

The raw data produced from either a single-fault- or multi-fault-workload availability benchmark is rather unwieldy, and therefore some standard techniques for analyzing and reporting it are required.

The simplest way to handle the data from the runs with fault injection is to plot it graphically, with the quality of service metrics on the vertical axis and time on the horizontal axis, as in the example graph of Figure 1. The graph is then overlaid with confidence intervals calculated from the runs in which no faults were injected; these intervals indicate the range of quality of service values that are statistically normal. Finally, the times at which faults were injected are marked on the graph.

Notice that in Figure 1, the quality of service metric has been quantized by plotting the data as average values over a fixed time interval. This averaging is necessary for throughput-like quality of service metrics, which are non-instantaneous and have an inherent averaging window. Although instantaneous metrics like response time do not

require averaging to be plotted in our framework, averaging can still prove useful by smoothing out some of the inherent fine-timescale variability of such data. Of course, in any scheme where averaging is used, the size of the averaging window can significantly affect the results, and therefore great care must be taken in choosing the value of this parameter.[1] Where possible, the window should be chosen to be a natural quantity that is meaningful to the benchmark user, and not longer than the smallest quality-of-service deviation that is important to the user. As an example, a benchmark using one of the TPC database workloads would likely choose a one-minute averaging window for throughput data, since the natural metric for TPC workloads is transaction throughput per minute; if the user were interested in response time, he or she might limit the averaging to four seconds so as to be able to detect transactions over the four second response-time limit specified by TPC.

Graphs such as those in Figure 1 provide a good means by which the experimenter or system designer can study and understand the availability behavior of the system, and they are what we will use later in this paper to report our results for software RAID. In particular, the experimenter can use these graphs to focus on the points at which the measured values of the quality of service metrics fall outside the statistically normal range; these are the points where the system's availability has been compromised.

However, the graphs remain somewhat difficult to quantify and compare, especially if the benchmarks are to be used by end-users or customers. Several SPEC benchmarks do report graphs, and some customers do compare the graphs side-by-side. But we believe that the salient features of the graphs can also be distilled numerically, and we have identified an approach to doing so, although we have not tested it in practice. The idea behind the approach is to examine the quality of service curve for a particular experiment, identify all deviations from the statistically normal range, and then characterize—via mean, standard deviation, and possibly a distribution function—the distributions of the frequency of those deviations, the length of those deviations, and the severity (height) of those deviations. By characterizing the distribution rather than just averaging, this approach should preserve, for example, the distinction between the system that is down 2

---

1. Similarly, the averaging window size is one of the most important parameters to standardize should availability benchmarks like these ever be put to commercial use. If left undefined, the averaging window size offers an opportunity for unscrupulous benchmarkers to "game" the benchmarks to produce inaccurate or misleading results.

seconds every hour and the one that is down one day every month. Of course, these characterizations could be distilled further, for example by simply reporting the product of the average length and average severity of the deviations, although at this point the benchmark result would begin to lose much of its descriptive power.

2.2 Implementing the Methodology for Software RAID

In the previous section, we presented a general methodology for benchmarking system availability. In this section, we describe how we implemented that methodology for measuring the availability of the software RAID implementations provided by Linux, Solaris 7 Server, and Windows 2000 Server.

The availability guarantees of RAID-5 are straightforward [13]. A RAID-5 volume can tolerate a single disk failure without loss of data. After that first failure, the volume can continue to service requests in *degraded mode*, although I/Os tend to be more expensive due to the need to reconstruct data on-the-fly. A second disk failure renders the data on the volume inaccessible. Some RAID-5 implementations support spare disks, and can restore redundancy by rebuilding onto the spare after the first failure; during this reconstruction period, the volume will still be destroyed if a non-spare disk fails, although failure of the spare disk can be tolerated.

2.2.1 Fault injection environment

For the experiments in this paper, we chose to limit the fault injection to faults affecting the disks comprising the software RAID volume, as those are the primary hardware failure points in a software RAID system. Since we wanted to generate a range of different disk faults in a controlled manner, we rejected the simplistic fault-injection technique of pulling disks out of a live system. Instead, we replaced one of the SCSI disks in the software RAID volume with an *emulated disk*, a PC running special software with a special SCSI controller that makes the combination of PC+controller+software appear to other devices on the SCSI bus as a disk drive (*i.e.*, a SCSI target rather than a SCSI controller). Thus our systems under test saw the PC emulating the disk as a real disk drive.

Our emulated disk consisted of an AMD-K6-2-350 PC with an ASC ASC-U2W SCSI adaptor, running Windows NT with the ASC VirtualSCSI Target Mode Emulation

library installed [7]. We adapted the library to emulate one or two SCSI disk drives by converting I/O requests to the emulated disk into reads and writes to two large backing files on a dedicated local disk on the emulation machine. The dedicated disk was an IBM DMVS18D 18GB 10000RPM Ultra2-LVD SCSI drive with an NTFS file system placed at the start of the disk. The files holding the contents of the emulated disks were the only files on the local disk, only one file/emulated disk was active at once in any given experiment, and all accesses to the backing files passed through the NTFS file system layer but bypassed the buffer cache. The emulation layer added a constant overhead of approximately 510 microseconds to each disk I/O, as measured by the Skippy disk characterization benchmark [47]. Compared to a Linux file system on one of the real disks used in our RAIDs, this emulation overhead translates to 10% fewer seeks per second, 41% less write bandwidth, and 16% less read bandwidth, as measured by the 100MB Bonnie benchmark.

We modified the disk emulator to allow the injection of faults into the emulated disk. To make our benchmarks as realistic as possible, it was essential that our set of injected disk faults closely match the types of disk faults seen in practice. To that end, we turned to a study performed as part of the aforementioned Tertiary Disk project at UC Berkeley. Using the 368 disks in the TD array, Talagala recorded the types of faults that occurred over an 18-month period [47]. She found that the most common errors and failures affecting disks included recovered (media) errors, write failures, hardware errors (such as device diagnostic failures), SCSI timeouts, and SCSI bus-level parity errors.

Using this set of errors as a guide, we selected several categories of faults to include in our emulator:

- *correctable media errors* on reads and writes, to simulate disk sectors starting to go bad;
- *uncorrectable media errors* on reads and writes, to simulate unrecoverably-damaged disk sectors;
- *hardware errors* on any SCSI command, to simulate firmware or mechanical errors;
- *parity errors* at the SCSI command level, to simulate SCSI bus problems;
- *power failures* that simulated a disk being disconnected, both during and between SCSI commands;

- *disk hangs* that simulated disk firmware bugs/failures both during and between SCSI commands (these appear as SCSI timeouts to the controller).

All of the faults (except for the fatal ones, like simulating disk power down or infinite timeout) could be inserted either in transient mode, in which case they appeared once then disappeared, or in sticky mode, in which case they continued to manifest themselves once injected. We were particularly interested in the behavior of the software RAID systems in response to the transient faults, as results from Talagala's TD study indicate that disks rarely fail fast, but rather tend to die slowly with an ever-increasing number of transient and correctable faults [47]. Most availability guarantees made by RAID systems speak only of discrete failures, not of such "fail-slow" failures.

As desired, our set of injectable faults closely matches the set of error conditions seen in the TD array. Note that we were unable to inject one of these types of error condition with our fault-injection harness: the SCSI parity errors at the level of the SCSI electrical protocol. Simulating this type of fault requires either direct access to the wires of the SCSI bus or to low-level registers within the controller, neither of which were available to us.

2.2.2 Configuration of systems under test

We examined three software RAID implementations in our experiments, those shipped with Linux, Solaris 7 Server with Solstice DiskSuite, and Windows 2000 Server. In all cases, the OS and RAID system were installed on a the same test system; the complete details of our test environment are listed in Figure 2. Note that each of the three physical RAID drives in the test system had a private fast/wide SCSI bus that was not shared with any other device. A 1GB partition was created at the beginning of each physical drive for use in the experiments; the remainder of the space on each drive was unused. The emulated disk (*i.e.*, the PC running the emulation software) was also connected to a dedicated SCSI bus on the machine under test. Two 1GB emulated disks were created; one was used in the RAID and the other was left as a spare (thus the two were never simultaneously part of the active RAID volume).

The three systems were configured as web servers with the documents served from the RAID volume and the logs written to the RAID volume. We wanted to select the web server that would be typically used with each OS, so we chose Apache for the Linux and

Configuration Parameter		Linux	Windows	Solaris
Test Platform	CPU	AMD K6-2, 333 MHz		
	Memory	64 MB, 66 MHz ECC SDRAM		
	System disk	Seagate 5400 RPM IDE		
	Physical RAID disks	IBM DMVS18D: 18 GB, 10000 RPM, Ultra2-LVD SCSI		
	SCSI busses	4 Fast/Wide (20 MB/s) SCSI		
	SCSI controllers, physical disks	Adaptec 2940UW, Adaptec 3940W		
	SCSI controller, emulated disks	Adaptec 2940UW		
Software Configuration	OS version	RedHat 6.0	Windows 2000 Server, RC build 2128	Solaris 7 for Intel Architectures
	RAID software	raidtools-0.90-3	<included>	Solstice DiskSuite 4.2
	File system	ext2	NTFS	ufs
	File system parameters	4KB block, stripe width 8	<default>	<default>
RAID Configuration	RAID level	5	5	5
	RAID volume size	3 GB	3 GB	3 GB
	RAID disks, active	3 real, 1 emulated	3 real, 1 emulated	3 real, 1 emulated
	RAID disks, spare	1 emulated	1 emulated	1 emulated
	Hot spare?	Yes	No	Yes
	RAID parameters	left-symmetric parity, chunk size 32	<default>	<default>
Web Server Configuration	Web server	Apache 1.3.9	IIS 5.0	Apache 1.3.9
	Web server parameters	<default>	"More than 100,000 hits/day"	<default>

Figure 2: Configuration of test environment for RAID experiments. The same test platform was used for all three RAID systems. All system parameters were left at default values except where noted; non-default parameters were used only when no defaults were supplied or when documentation suggested otherwise.

Solaris systems, and Microsoft Internet Information Server (IIS) for the Windows 2000 system. Other than relocating the logs and document directories to the RAID volume, the servers were left in their default configurations; the details of our web server configurations are also listed in Figure 2.

2.2.3 Workload generator and data collector

In order to complete our experimental testbed, we needed a source of workload for the web servers running on each OS, and a means of continuously measuring the quality of service delivered by the web servers over time. We chose to use SPECWeb99 [45], a standard web performance benchmark, for both of these tasks. SPECWeb99 uses one or more clients to generate a realistic, statistically reproducible web workload; its workload models what might be seen on a busy major server, and includes static and dynamic content, form

submissions, and server-side banner-ad rotation. In each iteration, the benchmark applies a load designed to elicit a certain aggregate bandwidth from the server, then measures the percentage of that bandwidth that was actually achieved. It also measures the number of hits per second delivered by the server and the average response time; we chose to use the number of hits per second (a throughput-oriented performance metric) as the quality of service metric as it was the most tractable and because the other metrics tracked it relatively closely.

We modified the workload generator slightly so that it would fit our model of continuous performance measurement over time: we removed all warm-up and cool-down periods other than the initial warm-up period, adjusted the per-iteration time to two minutes, and set the number of iterations to a very large number (manually stopping the generator when the benchmark was complete). These adjustments allowed us to obtain performance measurements every two minutes, with each number reflecting the average performance over the previous two-minute period. We chose a two minute measurement interval as a compromise value: longer intervals have the drawback of obscuring the system's dynamic behavior, whereas results obtained from shorter intervals can be confounded by natural fluctuations in the applied workload.

We also adjusted the workload generator to reduce the amount of dynamic content from 30% to 1% to keep the disks busy and to avoid saturating the CPU. This restriction was necessary because we used the default high-overhead `perl-cgi` implementation for dynamic content and the CPU on our server testbed was not able to keep up with the higher level of dynamic content.

We configured the applied workload to be just short of the saturation point on each of the three systems by increasing the number of active connections per second (the SPECWeb99 load unit) until a knee was observed in the performance curve, then backing off the load by 5 connections per second. The three systems each saturated at different points, and thus we applied a different level of load to each system in our tests; this accounts for the differences in absolute performance that show up in Figures 4 and 5, below. We chose this load profile instead of applying a consistent load to all three machines in order to isolate the worst-case availability impact on each system. This profile also ensures that we were making fair comparisons between the systems, as some availabil-

ity behavior (such as RAID reconstruction speed) can be affected by the amount of free system resources. Where pertinent, we also discuss results from experiments in which the applied load was reduced to below the saturation point on each system.

Finally, note that we observed heavy disk activity during the benchmark runs on all three systems, indicating that server-side caching effects were not significant.

2.3 Results

In this section, we present the results of applying our availability benchmarking methodology to the software RAID implementations provided by Linux, Solaris, and Windows 2000. We first look at the single-fault availability microbenchmarks, then move on to study more complex multi-fault availability macrobenchmarks.

2.3.1 Single-fault microbenchmarks

Recall that single-fault microbenchmarks involve injecting a single fault into a running system and observing the resulting behavior of that system without any human intervention. To perform these microbenchmarks for the software RAID systems, we first configured the RAID volume to its nominal state: all disks working, and all spares available. We then started the SPECWeb99 workload generator and allowed it to reach steady state. We next injected a single fault, and allowed the system to continue running (collecting performance data) until the system recovered (performance returned to its steady-state level), stabilized at a different performance level than its steady-state level, or crashed. We define a system crash as the *system* failing to provide service (zero web hits per second) for at least 20 minutes with no apparent signs of ever returning to service.

In all cases, the faults that we injected were chosen to affect active disk blocks, guaranteeing that the system would be aware of them. By doing so, we avoid injecting so-called *latent* faults, faults that cannot cause failures since they affect only unused data or control paths. We feel this is a reasonable policy for an availability microbenchmark, as the goal of such benchmarks is to characterize the system's response to specific faults, and not to measure susceptibility to randomly-placed faults.

We injected a total of 15 types of faults, listed in Figure 3. Each fault-injection experiment was repeated at least twice, and in all cases, similar behavior was observed in each

Type of Fault	Behavior		
	Linux	Solaris	Windows
Correctable read, transient	C-1	A	A
Correctable read, sticky	C-1	A	A
Uncorrectable read, transient	C-1	A	A
Uncorrectable read, sticky	C-1	C-2	B
Correctable write, transient	C-1	A	A
Correctable write, sticky	C-1	A	A
Uncorrectable write, transient	C-1	A	B
Uncorrectable write, sticky	C-1	C-2	B
Hardware error, transient	C-1	A	A
Illegal command, transient	C-1	C-2	A
Disk hang on read	D	D	D
Disk hang on write	D	D	D
Disk hang, not on a command	D	D	D
Power failure during command	C-1	C-2	B
Physical removal of active disk	C-1	C-2	B

Figure 3: Classification of system behavior for each of the injected faults. The letters in the rightmost three columns correspond to the pattern of behavior observed after the specified fault is injected; the behaviors are discussed in the text below, and illustrated graphically in Figure 4.

iteration. In our experiments, we found no evidence of corruption from any injected fault. All faults that could potentially result in corrupted data were either detected by the OS's disk driver or RAID layer. What differentiates the systems is not their detection abilities, but their behavior in response to the detected faults.

Surprisingly, these response behaviors across the three systems and the 15 types of injected faults can be classified into only five distinct categories, also listed in Figure 3. Representative availability graphs for each of these categories are plotted in Figure 4. We classify two of the behaviors (**C-1** and **C-2**) as subcategories of the same major behavior category, as they represent the same response behavior (automatic reconstruction) but differ in their performance characteristics. Note that each graph in Figure 4 plots the change in two metrics with respect to time. The first metric, represented by a solid line, is the same performance metric discussed above: the number of hits per second delivered by the web server running on the system under test, averaged over two-minute intervals. The second metric, represented by a broken line, represents the minimum number of disk failures the system is theoretically able to tolerate; it is effectively a measure of the system's data
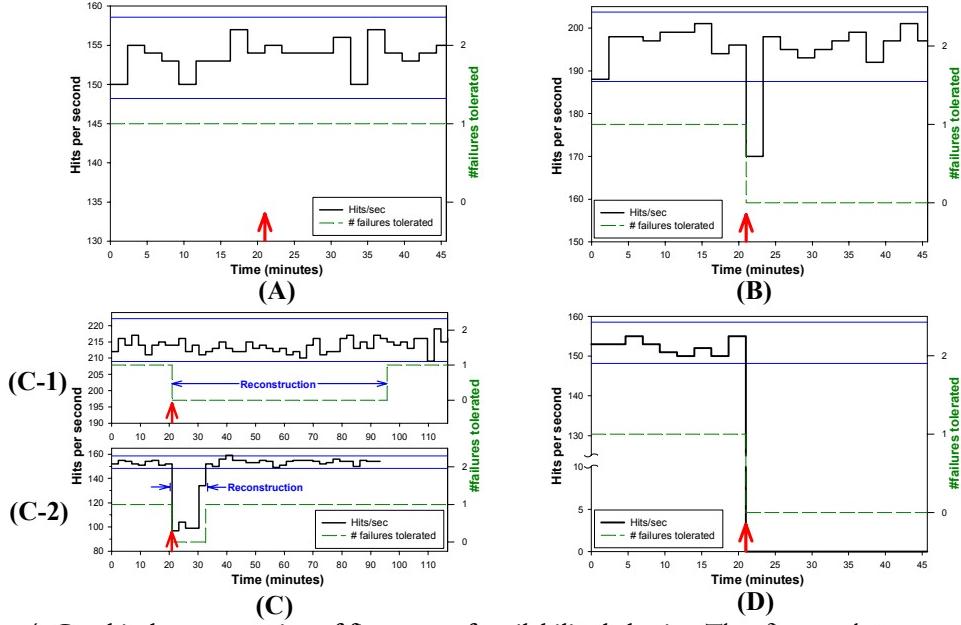
Figure 4: Graphical representation of five types of availability behavior. The figure plots representative availability graphs displaying the five different patterns of behavior observed after injecting faults into the three software RAID systems. Each graph plots two metrics: on the left vertical axis, and represented by a solid line, is the number of hits per second sustained by the web server on the system under test, reported as a single average value over each two-minute interval. On the right vertical axis, and represented by a broken line, is the theoretical minimum number of disk failures the system should be able to tolerate without losing data. Fault injection points are represented by heavy arrows, and 99% confidence intervals for the normal (non-faulty) behavior of the systems are defined by the thin horizontal lines. Figure 3 maps each type of injected fault into one of these five behaviors (A, B, C-1, C-2, D) for Linux, Solaris, and Windows 2000.

redundancy. Note that the graphs also show 99% confidence intervals that were computed from the traces of the systems' normal no-fault performance.[2]

Of the four major categories of observed behavior, the first, **A** in Figure 4, represents the behavior pattern that occurs when an injected fault has no effect on the RAID system. This graph plots the behavior of the Solaris system in response to a transient, correctable read fault. Notice that the performance curve remains within the confidence intervals despite the injection of the fault; the redundancy measure remains unchanged as well. Effectively, the Solaris system ignores this fault, as it is essentially benign; the disk correctly satisfied the read request, but needed to use ECC bits or multiple reads to obtain the data. Both the Solaris and Windows 2000 systems displayed behavior of this type. Solaris responded this way to almost all non-fatal faults that we injected, including transient uncorrectable faults (such as a transient, non-repeatable write failure); the one exception was a transient illegal command fault, a case that we discuss further in the analysis

---

2. Analysis showed that the no-fault performance data was normally distributed; thus, the 99% confidence intervals were computed as 2.576 sample standard deviations on either side of the sample mean.

section, below. Windows 2000 behaved similarly to Solaris, although it was slightly less tolerant of write errors: it exhibited pattern **B** rather than **A** for a transient uncorrectable write fault, indicating that the affected disk would be considered failed. In no cases did Linux exhibit pattern **A**—it never transparently tolerated a non-fatal fault.

The second category, **B** in Figure 4, is more complicated. In this case, the fault is severe enough that the RAID system stops using the affected disk, but is not so severe that the RAID system cannot tolerate it. The performance is slightly affected only during the interval in which the fault was injected, as the system detects and recovers from the fault. The redundancy curve indicates that the faulty disk is no longer used: in this case, the system does not automatically rebuild onto a spare disk, and thus the system cannot tolerate any more disk failures. The particular data plotted in Figure 4(B) is the behavior of Windows 2000 in response to a simulated power failure on one disk of the array (equivalent to physically pulling an active drive from a hot-swap array). This pattern also characterizes Windows's response to other severe faults, including sticky uncorrectable read faults and all uncorrectable write faults.

The magnitude of the performance drop during the fault-injection iteration depended on the type of fault; for uncorrectable writes, it was about 4% of the mean performance, and for power failures, it was about 13% of the mean. Note that the performance drop during the fault-injection iteration occurs because the server is near saturation. If we reduce the applied load by just over 20%, the observed performance drops become statistically insignificant. This indicates that Windows is able to trade spare resources for reduced availability impact in certain failure scenarios.

Neither Solaris nor Linux exhibited pattern **B**, as they both support automatic recovery onto a spare disk: when the Solaris or Linux software RAID driver detects a fault severe enough to stop using a disk, it immediately begins reconstructing the data from the failed disk onto the available hot spare. This pattern is illustrated in the graphs labeled **C-1** and **C-2** in Figure 4. **C-1** plots Linux's response to a transient correctable read fault, and **C-2** plots Solaris's response to a sticky uncorrectable write error.

In the Solaris case, we see that the performance curve drops significantly below the lower bound of the confidence interval during the reconstruction period. In contrast, Linux's performance during its entire reconstruction period is statistically indistinguish-

able from its unperturbed performance. However, Solaris completes reconstruction significantly faster than Linux. The significance of these behavioral differences will be discussed further when we compare the reconstruction behavior of Solaris and Linux with Windows's non-automatic reconstruction in Section 2.3.2.

Note that during reconstruction, the redundancy curve is not well-defined; the system cannot tolerate a fault to any of the data disks, but it can tolerate a fault to the spare (the destination of the reconstruction).

While Solaris exhibited its version of pattern **C** only for three of the 15 faults (two of which were unquestionably fatal faults), Linux exhibited pattern **C-1** for every injected fault but those falling into pattern **D** even if the fault was transient and non-fatal (like a correctable read).

Finally, the last category, **D**, represents what happens when the RAID system is unable to tolerate the injected fault. As can be seen, the performance drops to zero when the fault is injected; this is usually a result of the RAID driver or operating system hanging. The redundancy curve is not well-defined in this case, since the system is not operational. We observed this type of fault in Solaris, Linux, and Windows when we injected particularly pathological disk hangs in the middle of SCSI command execution, for example simulating a drive power failure or shutdown during command processing. While we expect that these kinds of failures can occur in practice, note that no such failures were observed in the Tertiary Disk study.

Analysis. Although limited to a single fault each, these microbenchmark results reveal interesting facts about the availability guarantees of Linux, Solaris, and Windows 2000; none of these facts were stated in the documentation supplied with the three systems. Most illuminating are the conclusions that can be drawn about how the three systems treat transient faults. If we exclude the pathological disk hangs and power-failure faults, 8 of the remaining 10 injected fault types simulate transient or recoverable errors that in isolation do not indicate immediate disk failure. Four of these 8 do not even require that the corresponding I/O's be retried. The remaining two faults (sticky, uncorrectable reads and writes) are the only faults in the set of 10 that indicate that the disk is in an unrecoverable state.

Yet for every fault in this set of 10 non-pathological faults, the Linux system exhibited behavior of type **C**, in which the faulty disk is immediately removed from service. In contrast, both Solaris and Windows kept the faulty disk in service on 7 of the 10 non-pathological faults (*i.e.*, 7 of the 8 recoverable errors). Solaris disabled the faulty disk (pattern **C-2**) upon the two unrecoverable faults (sticky uncorrectable reads/writes) as well as on a transient illegal command fault. This behavior is arguably slightly more robust than that of Windows, which disabled the faulty disk (pattern **B**) upon the two unrecoverable errors and a transient uncorrectable write, since an illegal command error typically implies a coding error in the driver or a serious disk firmware error, rather than a potentially transient magnetics glitch.

From these observations, we can conclude that Linux's software RAID implementation takes a totally opposite approach to the management of transient faults than do the RAID implementations in Solaris and Windows. The Linux implementation is paranoid—it would rather shut down a disk in a controlled manner at the first error, rather than wait to see if the error is transient. In contrast, Solaris and Windows are more forgiving—they ignore most transient faults with the expectation that they will not recur. Thus these systems are substantially more robust to transients than the Linux system. Note that both Windows and Solaris do log the transient errors to varying extents, ensuring that the errors are reported even if not acted upon. Windows is more explicit with its reporting, for example visually flagging a disk as "at risk" in the RAID management GUI upon a correctable write error, whereas Solaris relies on the system log for its error recording.

We cannot draw conclusions about a RAID system's overall robustness based solely on its transient-error-handling policy, however. There is another factor that interacts with a system's error handling, and that is its policy for reconstruction. The microbenchmarks demonstrate that both Linux and Solaris initiate automatic reconstruction of the RAID volume onto a hot spare when an active disk is taken out of service due to a failure. Although Windows supports RAID reconstruction, the reconstruction must be initiated manually, as discussed further in Section 2.3.2, below. Thus without human intervention, a Windows system will not rebuild redundancy after a first failure, and will remain susceptible to a second failure indefinitely.

The policy choice of automatically or manually-initiated reconstruction interacts strongly with the transient error-handling policy in affecting system robustness. A paranoid RAID implementation without hot spares is very fragile, as it takes only two transient errors to corrupt the RAID volume; likewise, a more forgiving RAID implementation has less of a need for hot spares as it will only stop using a disk upon a serious fault. Thus in our case, the non-robustness of the Linux implementation's paranoid approach to transients is mitigated somewhat by its automatic reconstruction, and similarly Windows's lack of automatic reconstruction is partially mitigated by its robustness to transients. Solaris seems to combine the best of both: robustness to transients plus automatic reconstruction upon a fatal error.

Returning to the three systems' transient error policies, if we consider these policies in the context of real failure data, such as that gathered by the Tertiary Disk project, it is clear that none of the observed policies is particularly good, regardless of reconstruction behavior. Talagala reports that transient SCSI errors are frequent in a large system such as the 368-disk Tertiary Disk farm, yet rarely do they indicate that a disk has truly failed [47]. Tertiary Disk logs covering 368 disks for 11 months indicate that 13 disks reported transient hardware errors, yet only two actually required replacement. Those two did not "fail-fast" with head crashes, either: both were replaced due to an excessively large number of transient errors. Additionally, due to the effect of shared SCSI busses and at-times flaky SCSI cabling, at some point over that period every disk in the system was involved in some sort of SCSI error (such as a parity error or timeout) [48]. Even if we ignore these SCSI errors and focus only on the transient hardware errors, Linux's policy would have incorrectly wasted 11 real disks (3% of the array) and potentially 11 spares (another 3% of the array) due to its over-zealous reaction to transient errors. Even worse, if the array did not have enough spares to keep up with the disk turnover, data could have been lost despite the fact that no disk truly failed. The response of Solaris or Windows 2000 would also have been less than ideal, as these systems most likely would have ignored the stream of intermittent transient errors from the two truly defective disks, requiring administrator intervention to take them offline.

A better RAID implementation would have a more balanced policy for dealing with transient errors. For example, it might be less paranoid initially, tolerating transient faults

until they reached a certain frequency or absolute count, at which point the system would declare a disk dead and stop using it (note that our macrobenchmark experiments showed that neither Windows nor Solaris did this). This kind of policy balances the need for long-term availability (which favors a more relaxed policy) with the fact that disks tend to fail with a stream of transient errors rather than failing fast.

Although none of the RAID implementations we examined is ideal, we can conclude from the microbenchmarks that either Solaris's or Windows 2000's RAID is more suitable for applications requiring high long-term data availability, as both are less likely to fall prey to multiple transient errors (especially in systems that are not closely monitored or conscientiously administered). However, in certain situations, the Linux implementation would still be a reasonable choice. For example, if application performance is most important, transient errors are expected to be infrequent, and repair times are short (*e.g.*, in systems with high-quality, well-administered hardware), then Linux's more paranoid policies may be appropriate. There is a tradeoff that must be carefully evaluated, though: even if repairs can be made quickly, the increased frequency of repair in a Linux environment could lead to an overall reduction in system availability as a result of introducing greater risk of operator-induced failures during maintenance. We show in Section 3.3 that such risks are far from negligible, especially for Linux.

Our results and analysis also argue strongly for the importance of exposing the policy decisions that affect availability in systems like these software RAID implementations. Ideally, the policies would be made configurable, for example by allowing the administrator to select a point on the spectrum between Linux's paranoid response to transients and Solaris's tolerance of them. Doing so would make the policies explicit, and may even simplify maintenance of the system by increasing its predictability, thereby eliminating the need for the administrator to guess at how the system will behave under various conditions. Furthermore, while it can be argued that introducing extra tuning knobs does contribute to the system's complexity, the maintainability cost of initially setting the knob is likely to be far outweighed by the maintainability cost of discovering the cause of a performance degradation or of repairing the system after a double failure, either of which can happen if the knob is set incorrectly.
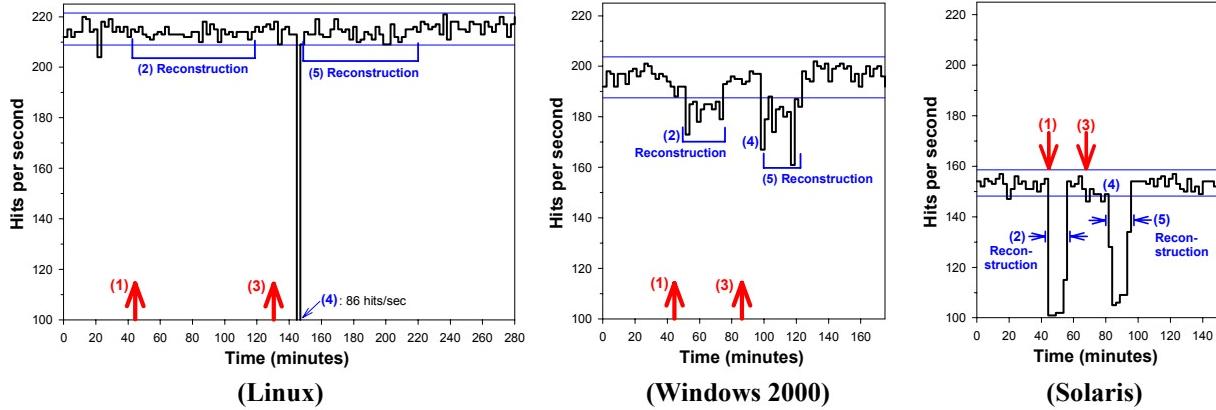
Figure 5: Availability graphs for an availability macrobenchmark with a multiple-fault workload. On the vertical axis, and represented by a solid line, is the number of hits per second sustained by the web server on the system under test. The change in this metric is plotted versus time on the horizontal axis. The thin horizontal lines represent the 99% confidence interval defining the system's normal (no-fault) behavior. The two injected faults are indicated by heavy arrows. The numbers in parentheses on each graph indicate the corresponding part of the fault scenario, as described in Section 2.3.2. The absolute performance differences between the three systems are due to different applied loads, as described in Section 2.2.3.

Even if they are not made configurable, availability policies such as those governing the system's response to transient errors should at the very least be documented so that administrators and buyers can evaluate the potential robustness of their systems in their particular environment. Until such documentation is commonplace, availability benchmarks such as those described here may well remain the only way to identify and evaluate these important but well-concealed policies.

2.3.2 Multiple-fault macrobenchmarks

After measuring the effects of single failures on the availability of the Linux, Solaris, and Windows software RAID implementations, we next constructed two fault workloads designed to mimic real-world scenarios and applied them to the three systems.

Scenario 1: Reconstruction. The first scenario includes five events, and models a situation in which a nominally-configured RAID-5 volume with one spare (1) experiences a failure on one of its active disks, (2) is reconstructed (automatically or manually) using the spare, and (3) later experiences a failure on the then-active spare. The scenario is finished by (4) the administrator replacing the two failed disks and (5) reconstructing the volume's redundant data onto one of the new disks. The behaviors of Linux, Windows 2000, and Solaris on this macrobenchmark are plotted in Figure 5. Note that for Windows, we inserted a 6-minute delay to simulate sysadmin response time between detecting the first

failure and manually starting the reconstruction. The process of "replacing" the broken (simulated) disks was performed manually, and took approximately 90 seconds in each case.

One obvious difference between the behaviors of the three systems on this benchmark is that Linux and Solaris automatically reconstruct whereas Windows requires human intervention. Most interesting is the difference in reconstruction time between the three systems, and in the performance impact of reconstruction in each case. Linux is the slowest to reconstruct the 1GB of missing data, taking well over an hour each time. However, there is no significant effect on application performance during reconstruction; other than during the time that the disks were being replaced, the performance curve does not fall outside of the confidence interval for normal behavior while reconstruction is taking place.

Solaris defines the opposite extreme. Its reconstruction is over 7 times faster than Linux's, lasting just over 10 minutes for 1GB of data. But this speedy reconstruction comes at a performance cost: the web server performance on Solaris is below the lower bound of its normal behavior for the entire reconstruction interval, with a maximum deviation of 34% from its mean no-fault performance.

Windows's behavior is similar to Solaris although not as extreme. Its reconstruction lasts approximately 23 minutes, over twice as slow as Solaris but still more than three times faster than Linux. Windows too shows a performance drop during reconstruction, but it is less significant than Solaris's: the worst-case performance observed was only about 18% below the no-fault mean.

From these observations we can conclude that Solaris and Windows are dedicating more disk bandwidth to reconstruction than is Linux. Our benchmarks have again revealed a hidden design tradeoff in the three systems: Linux chooses to emphasize preserving application performance over speedy reconstruction, even though it sacrifices short-term availability. In contrast, Solaris puts a high priority on restoring redundancy despite the performance impact. Windows makes the same tradeoff toward prioritizing reconstruction, but does so less aggressively than Solaris.

One might argue that these policy differences are irrelevant, since even our slowest measured reconstruction times (on Linux) are still short enough that they have little

potential impact on data availability; double faults are unlikely to occur with such little spacing between them. However, recall that these reconstruction times are for a 1GB disk in a 3GB RAID volume. These capacities are unrealistically small by today's standards. As single disk capacities head towards the 100GB mark, reconstruction times on systems like Linux threaten to scale to days and even weeks. When a system's window of vulnerability is this long, double faults (especially transients) become a real threat, and so slow reconstruction behavior may become a significant practical factor in a system's availability and reliability.

Another interesting characteristic of the RAID systems' reconstruction implementations is how reconstruction behavior changes as the load on the system is reduced. We found that at lower loads (such that the systems were unsaturated), Linux and Solaris each exhibited unchanged reconstruction behavior compared to the saturated case, in terms of both reconstruction time and performance impact. In contrast, Windows was able to decrease both its reconstruction time and the impact of reconstruction on application performance. Our hypothesis is that these behaviors are a function of the scheduling discipline in each of the OSs as well as the priority each system assigns to the reconstruction task. The implication of these behaviors is again significant for availability: Windows seems to be the only system of the three that is able to use the excess resources resulting from lower imposed load to mitigate the availability impact of reconstruction. In practice, this means that Windows is the only system of the three that can take advantage of hardware with a higher saturation point to improve its availability characteristics as well as its performance potential.

The differences in reconstruction philosophy revealed by this benchmark once again argue for the importance of exposing policies that affect availability. The three RAID systems examined here offer different robustness guarantees because of their undocumented reconstruction policies. We saw this above in how Windows compared to the other systems under reduced load. Another example is that Linux, with its slow, low-priority reconstruction, has a much larger window of vulnerability to double failures, a weakness exacerbated by its susceptibility to transient errors. This policy is unsuitable if data integrity is most important, and in that case a policy like Solaris's is a better choice. On the other hand, if delivering consistent application performance is more important than pre-

serving the data at all costs, then Linux's policy is reasonable and Solaris's unacceptable. An ideal system would offer the administrator a spectrum of choices between these two extreme policies, but we feel that every system should at least document its chosen policy. Benchmarks such as these offer a convenient tool for doing so.

Scenario 2: Double failure. The second scenario mimics a catastrophic failure in RAID systems reported anecdotally by multiple sources. The scenario begins when a nominally-configured RAID volume (1) experiences a disk failure that causes the faulty disk to be removed from service and (2) begins reconstruction (automatically or manually). At that point, (3) the well-meaning system administrator attempts to replace the failed disk, but accidentally pulls out the wrong disk—one of the remaining live disks rather than the dead one; (4) he or she then tries to restore the system to a working state. Removing the live disk should result in a catastrophic failure of the RAID volume, although it did not do so in all cases, as we discuss below. Note that the likelihood that (3) will occur depends on the maintainability of the RAID system; in Section 3 of this report, we will address this question directly. For now, we assume that step (3) has occurred, and consider the system's availability after that point. Up until (3), the availability graphs are relatively uninteresting, confirming the expected behavior, and are not reproduced here.

What is interesting is the behavior of the systems after the catastrophic failure in step (3), and the difficulty of restoring service on the system. We describe this behavior only qualitatively, since quantifying takes us into the realm of maintainability benchmarks, which are considered later in Section 3.

In this scenario, the last, "catastrophic" failure is actually reversible. According to the RAID availability semantics, the RAID volume should stop serving requests upon a double failure. If the RAID implementation queues writes to the removed disk while it is unavailable, the administrator could put the disk back in, and theoretically, the system should be able to recover. We tested this hypothesis on the three systems in order to see how close each of them came to this theoretical possibility.

Windows 2000 actually came remarkably close, although it does not queue writes to disconnected disks. After reactivating the accidentally-removed disk (which required a few GUI operations), Windows allowed the RAID volume to be accessed despite its possibly corrupt state, and the web server resumed serving requests to the SPECWeb99 clients.

Running CHKDSK as recommended revealed no file system corruption (probably due to the journaling nature of NTFS). Since the web workload was essentially read-only except for the log writes, the only data lost was logging information.

In contrast, we found it impossible to resurrect the Linux RAID volume. The tool used to reintegrate a disk into the volume seemed to only be capable of adding new disks to the volume as spares, which are then automatically used as the target of a reconstruction. There was no obvious way to use the existing tools to convince Linux that the replaced disk contained real data. Therefore, the only way to resurrect the volume was to recreate and reformat it, then restore data from backup.

Solaris demonstrated radically different behavior than the other two systems. Unlike the other systems, it did not disable the RAID volume after the double failure: it kept the array active with the two still-functioning disks and the partially-reconstructed spare. This behavior violates the availability semantics of RAID-5, since at this point a large portion of the data is missing (any data that had not yet been reconstructed on the spare is permanently lost). By keeping the RAID array active and using the nonsensical data on the partially-reconstructed spare, Solaris allows applications to read garbage data. In our case, this was manifested by the web server returning garbage to the SPECWeb client and via numerous UFS file system corruptions as reported by fsck. Furthermore, when we plugged the accidentally-removed disk back in, Solaris was happy to automatically switch back to using it to service I/Os, deactivating the partially-reconstructed spare. However, because Solaris had continued to use the array while the second disk was removed, the data on that disk was significantly out-of-date and the file system was corrupted as a result of reinserting it.

We believe that Solaris's behavior is absolutely incorrect for a RAID system, and have reported it to the developers of the DiskSuite component of Solaris. A RAID system should not fabricate data to maintain availability unless explicitly requested to do so, *i.e.*, by manually forcing the reactivation of a reinserted disk, as with Windows. Furthermore, we were not able to find any mention of this behavior in any of the Solaris documentation, which again argues for the importance of benchmarks like these to expose the undocumented availability policies in systems like these software RAIDs.

Thus in this scenario, Solaris clearly is the loser due to its willingness to transparently serve up garbage data. But Windows 2000 wins on maintainability, as its robust file system and flexible RAID implementation allows the opportunity for at least some use of the RAID volume to continue servicing user requests while the system is being restored from backup (but only at the explicit request of the administrator, unlike Solaris). Although this may not always be the best thing to do, Windows provides the ability should it be desired.

In this second macrobenchmark, we have the beginnings of a framework for a combined availability and *maintainability* benchmark—the fault injection workload for this scenario brings the system to a state in which maintenance is required. To complete the benchmark, we need to quantify the human cost of repairing the system, either via a quantitative model of human-initiated maintenance, or via human experiments in which the maintenance cost is directly measured. The next section of this report, Section 3, considers the second approach; it introduces a general methodology for experiment-based measurement of maintainability, and presents a case study that represents the continuation of this multiple-fault availability benchmark.

Section 3

Toward Benchmarks for Maintainability

In this section, we present some initial thoughts on a methodology for benchmarking system maintainability, and a proof-of-concept case study implementation of that methodology. Our approach is based on a combination of the fault-injection techniques described in Section 2 and an experimental approach that relies on the participation of human subjects. After describing our approach, we present and analyze experimental results that continue the case study of Section 2.

3.1 Challenges in Benchmarking Maintainability

The task of quantifying the maintainability of a computer system is a challenging one. Two challenges stand out as particularly important, and particularly difficult. The first is that a system's maintainability depends directly on the way that the system is used. Differences in workload, availability requirements, and administrative policies between installations can all lead to vastly different maintainability challenges for the same system.

The second challenge arises because maintainability is, at its core, a *human* metric, reflecting the interaction of a human administrator's behavior with a system's design. Any measure of maintainability needs to capture a wide range of human factors, including everything from the amount of time it takes to perform maintenance, to the usability of the system's administrative interface, to the psychological effects of complexity in a maintenance task, and to the user's propensity for causing errors. Furthermore, the importance and behavior of these factors can differ between users: given the same system, in all likeli-

hood two different administrators will run into different types of problems while maintaining the system, and will draw different judgements about the system's ease of maintenance. This last point reflects the inherent variability that arises when human factors are considered. While performance benchmarks often attempt to strip out any potential for human variability (for example, by removing "think time" in scripted tasks), the human factors are so important in measuring maintainability that their accompanying variability must be accepted and incorporated into the benchmark.

While these two challenges are difficult ones, we believe that they can be overcome, and we will now outline our ideas on how to do so. We believe that the first challenge, that of the dependence of maintainability on workload, can be addressed by applying the ideas of *application-specific benchmarking* [42]. The idea behind application-specific benchmarking is to characterize the system of interest along as many different axes as possible, producing a large set of results that capture all of the fundamental behaviors of the system and any important interactions between the behaviors. Once this system characterization has been obtained, the system's workload is characterized in a similar way, by identifying and rank-ordering the particular characterization axes (system behaviors) that are relevant to the application or workload at hand. When appropriately merged and distilled, these two *independent* characterizations produce an overall benchmark score that is relevant to both the system and the workload of interest. While application-specific benchmarking has to date only been applied to the performance domain [8] [51] [52], we believe it is an ideal approach to use for maintainability: the approach can provide overall maintainability scores for a variety of different workloads and environments, while using a base system characterization that can be derived independently of the workload and environment. Note that, in the context of maintainability, the term *application* refers more to the system's environment, workload, and site policies than to any particular piece of software.

The second challenge in benchmarking maintainability revolves around how to incorporate human factors into the benchmark metrics, techniques, and results. One way to address this challenge would be to build a model of how human administrators respond to system-generated stimuli while carrying out maintenance tasks, and to then apply this model to the various system under test. Unfortunately, this is an extremely difficult and

probably impossible task, one that essentially requires building an artificial intelligence that is able to handle the myriad unforeseen situations that arise during system maintenance. Indeed, if we were able to build this model, it would represent a huge step forward in the state of the art of system maintenance, as the model could be used as a replacement for human administrators, instantly turning existing systems into futuristic self-maintaining entities.

A much more practical approach to addressing the human challenge is to simply incorporate human beings into the benchmarking process. Doing so incontrovertibly exposes the human factors involved in the maintenance of a test system. However, using human subjects has several implications on benchmark design. First, it makes experiments much more labor-intensive due to the need to train, evaluate, and debrief human subjects. More importantly, it requires extra care in choosing metrics and measurement techniques that can capture the impact of human factors while not being confounded by the inherent variability in behavior both across and within different human subjects. As we will show in the remainder of this section, these are not insurmountable problems, although they do influence the complexity and practicality of large-scale deployment and use of maintainability benchmarks.

3.2 A Proposed Methodology for Benchmarking Maintainability

As suggested by the discussion in the previous section, we propose a methodology for benchmarking maintainability that takes an application-specific approach, relying on experiments using human subjects to build the system characterization required by such an approach. From our discussion above, we know that an application-specific benchmarking approach has three main components: characterization of the system, characterization of the workload, and distillation of those characterizations into a single metric. In this report, we will confine our focus primarily to the first of those three components, system characterization, although we discuss our thoughts on the remaining two components as future directions, in Section 3.5.1.

Recall that the goal of system characterization is to collect a set of results that capture a system's behavior along as many different behavioral axes as possible. To do this, we must carry out three main tasks: defining the behavioral axes of interest, defining metrics

for quantifying a system's position along those axes, and finally measuring the values of those metrics on a the system of interest. We consider each of those tasks in turn in the following sections.

3.2.1 Defining characterization axes

The problem of defining characterization axes for a maintainability benchmark can be seen as equivalent to defining what exactly is encompassed by the term "maintainability". If we take maintainability as a measure of the cost or difficulty of system maintenance, we can divide the maintainability space along axes that correspond to particular *tasks* that are carried out in the course of system maintenance. Each possible *maintenance task* corresponds to one axis of the maintainability space, and a system is characterized in that space by measuring the cost of all of the appropriate maintenance tasks.

Of course, the space of possible maintenance tasks is quite large, running the gamut from the open-ended (like failure diagnosis and emergency hardware repair) to well-specified (like planned backups, software installations, or upgrades). Completely characterizing a system along all possible axes involves enumerating and characterizing all of these possible maintenance tasks, which is a daunting challenge. Therefore, we expect our benchmark methodology to used primarily for studying the maintainability impact of a few critical tasks, at least initially; this is the approach that we have taken in our case study. In Section 3.5.1, we discuss some ideas on how to manage the complexity of enumerating and measuring the full space of maintenance tasks, a Herculean undertaking that seems to be required if a fully-general application-specific maintainability benchmark is desired.

3.2.2 Metrics for the cost of a maintenance task

Having identified the axes of the maintainability space as individual maintenance tasks, we now consider appropriate metrics for positioning a system along those axes. In essence, we want to quantify the *cost* of a maintainability task, but doing so is complicated since we have no standard currency for evaluating that cost. We also have the problem that the cost can depend on human factors, and thus has some inherent variability.

We address the issue of defining maintainability cost by splitting it into three components, each of which characterizes a different part of the overall cost:

1. *TIME*: the minimum time that an administrator must spend on the task, assuming optimum conditions and excluding think time and any delays in the administrator's response to system output,

2. *IMPACT*: the minimum impact on system availability that results from performing the task in an optimum manner, as measured by an availability benchmark like those of Section 2,

3. *LEARNING CURVE*: how difficult is it for an administrator to reach the minimum time and impact measured by metrics #1 and #2.

In dividing up cost into these three metrics, we isolate as much as possible the system factors from the human factors. The first two metrics, TIME and IMPACT, are properties of the system itself; this is why we have defined the metrics in terms of optimal administrator performance. All human-related delay, fallibility, variability, and error-proneness is captured by the third metric, LEARNING CURVE; this metric measures primarily human factors, but still incorporates some system dependence, as the human factors arise from interactions between the human administrator and the system.

The three metrics warrant further discussion to elucidate some of their subtleties. We begin with the TIME metric. The intent of this metric is to capture the minimum amount of time that an administrator must spend with the system to perform a certain task; it is a measure of how efficiently a system uses an administrator's time. A system that requires constant "babysitting" should therefore have a worse (larger) TIME cost than a system that performs the same task while only requiring administrator involvement at the start of the task. To capture this distinction, we want our TIME metric to include only the time that the administrator is actively interacting with the system.

The subtlety arises when the administrator has to wait for the system in the course of carrying out a maintenance task. If the wait time is short, the administrator will probably sit it out, and therefore it should contribute to the TIME metric. If the wait time is long, however, the administrator will probably only wait a limited amount of time before switching to another task. Therefore, we would like to include waiting time in the TIME

metric only if it is short enough that the administrator cannot effectively switch to a different task in the meantime. As an example, consider the task of disk repair in a RAID system. During this task, the administrator probably will wait for a fast process to complete (such as rewriting the new disk's partition table), but probably will not stand by waiting for a long task like array reconstruction to complete. In this case, the TIME metric would include the partitioning delay but not the reconstruction delay. One way to handle this subtlety more formally is to only include waiting times that are shorter than some upper time bound; in our experiments, we take this approach with an arbitrarily-chosen value of 1 minute for that upper bound. Further study of typical administrator attention spans is needed to deduce a realistic value for the time bound.

We turn now to the IMPACT metric, and again find another subtlety. The intent of the IMPACT metric is to capture the degree to which system maintenance perturbs the normal operation of the system; this is an important factor in maintainability, as it can affect both the frequency at which maintenance is performed and the amount of pressure put on the administrator performing the maintenance. For example, a software patch is less likely to be installed if requires the shutdown of a critical system; the delay in application of the upgrade could potentially reduce the system's reliability or increase its security vulnerability.

The subtlety of the IMPACT metric concerns the definition of *normal operation*. If the maintenance task is designed to change the normal level of operation (for example, by replacing a slow CPU with a fast one), then the measured availability impact of the task will include both the impact of the task *procedure* as well as the impact of the *results* of the task. We want to include only the availability impact of the task procedure in the IMPACT metric. In the CPU-upgrade example, the new CPU might increase the system's baseline performance availability, but the upgrade procedure might require a system shutdown, temporarily dropping the system's availability to zero. It is the effect of the shutdown that we want to capture in the IMPACT metric, not the potential speedup from the new CPU.

Finally, we consider the LEARNING CURVE metric. This is a complicated metric to define quantitatively, although its intent is simple: the TIME and IMPACT metrics specify the minimum time cost and availability impact of a maintenance task, assuming optimal

behavior of the human administrator; the LEARNING CURVE metric should measure how difficult it is for an administrator to reach those points of optimality.

Although we have not yet formalized a precise definition of LEARNING CURVE, we can identify several factors that it must capture. First, it must characterize how quickly an administrator approaches the point of optimality in terms of task time, assuming that the task is carried out repeatedly and consecutively. The rate at which the administrator's time approaches the optimal TIME value and the total number of task repetitions needed to reach that point are both metrics that can be used. Second, LEARNING CURVE must characterize the administrator's proneness to errors during the learning process, a factor that speaks to the robustness and simplicity of the system. This factor also relates to the IMPACT metric: administrator errors can affect the system's availability, and thus the measured IMPACT value represents an availability bound that the administrator will approach as he or she gains familiarity with the system.

Finally, the LEARNING CURVE metric must capture the administrator's ability to retain what was learned—in particular, the effect of time away from the system on the administrator's completion time and errors committed for a particular task. One way to quantify this is to measure how poorly the administrator performs after being away from the system for some length of time, and how quickly he or she returns to the optimal time. We believe this final factor will highlight the differences between systems with complex, hard-to-retain interfaces and those with more intuitive, guided interfaces. All of these factors rely on the assumption that an administrator's behavior in carrying out a maintenance task will approach optimal as that task is carried out repeatedly and consecutively; in other words, we assume that the administrator will learn. Our experiments show that this is a reasonable assumption.

3.2.3 Measuring the maintainability cost metrics

Now that we have a set of metrics that can quantify the maintainability cost of particular task (and thereby quantify a system's space along each axis of the maintainability space), we must develop procedures for measuring those metrics on real systems. Measuring the first two metrics, TIME and IMPACT, is relatively straightforward.

The TIME metric can be measured in two ways: either directly from the system, or from data collected during human experiments. In the first case, the experimenter directly analyzes the system, measuring the minimum interaction times for each step of the maintenance task, and summing those up along with any waiting times that meet the duration-cutoff criterion described above. In the human experiment approach, the experimenter runs trials in which human subjects repeatedly perform the maintenance task until their times reach a plateau, indicating that they have reached the bottom of the learning curve. The best times across the subjects are then used to calculate the minimum TIME cost for the maintenance task. This is the approach we will take in our experiments. Finally, a hybrid approach is also possible, in which the experimenter first videotapes a single trained administrator correctly performing the procedure, then extracts and sums each of the interaction times and appropriate waiting periods from the videotape, excluding observed think time.

Measuring the second cost metric, IMPACT, boils down to performing an availability microbenchmark of the form described in Section 2. In such a benchmark, the maintenance task itself would comprise the fault injection workload; the final result graph would have to be distilled into a numerical value for easy representation as a characterization of maintainability cost.

In contrast to the other two metrics, measuring the LEARNING CURVE cost metric is more challenging due to the need to capture human factors and their inherent variability. We present here only our initial ideas; a great deal more thought and experimentation will be required to develop a more robust and practical approach.

Recall that we identified three factors that could contribute to LEARNING CURVE: time to reach optimality, errors made while reaching optimality, and retention. One way to measure the time it takes an administrator to reach optimality is to have him or her repeat the maintenance task repeatedly. We believe (and our experiments confirm) that as the administrator becomes more familiar with the task, he or she will reach a plateau where the time to perform the task will approximate the value of the TIME metric. The number of iterations required to reach the plateau can then be used to quantify the time-to-optimality factor. While this repetition procedure is taking place, we can record and classify the errors made by the administrator, giving us a measure of the second factor,

errors made while reaching optimality. Finally, to measure retention, we can use the following procedure. First, the administrator is allowed to reach the optimal point through the repetition procedure described above. Then, after some length of time during which the administrator does not perform the maintenance task, he or she is brought back and the repetition procedure is carried out again. This time, one would expect the administrator to both start out with a better time, and to more quickly reach optimality with fewer errors. Measuring the extent to which this happens provides a quantification of how well the administrator has retained his or her knowledge and understanding of the system over a given time period.

We have described our proposed approaches for measuring LEARNING CURVE as if they were carried out with a single administrator. In order to discover the effect of human variability, we in fact need to carry out these procedures with multiple administrators, and under various levels of time pressure and stress. However, the complexity of designing appropriate human subjects experiments to quantitatively characterize LEARNING CURVE is beyond the scope of this work; in our case study, we use a small-scale pilot study and evaluate LEARNING CURVE only in a qualitative manner.

3.3 Initial Experience: Continuation of the Software RAID Case Study

In the previous section, we have defined a methodology that should be capable of building a maintainability characterization of a system by evaluating the maintainability cost of its most important maintenance tasks. In doing so, however, we have defined a set of new and complex metrics that rely on novel measurement techniques involving human experiments. Because these metrics and techniques are so different than those traditionally used for performance benchmarking, we believe that it is necessary to explore their practical viability before pushing the methodology any further along the path to a complete application-specific maintainability benchmark. To this end, we have carried out a continuation of the software RAID case study described in Sections 2.2 and 2.3, this time focusing on maintainability rather than availability.

We had one main goal for this experimental work: to experimentally measure the maintainability costs for a simple maintenance task and compare those costs across systems. Continuing with the software RAID example, we selected the task of detecting and

replacing failed disks in a software RAID volume on each of three systems (the same Linux, Windows 2000 Server, and Solaris 7 Server systems as in the availability study). We chose to focus on the TIME and LEARNING CURVE costs, since the IMPACT cost is an availability cost that we already looked at in the context of our first multiple-fault availability macrobenchmark (see Section 2.3.2). As a side effect of designing and carrying out the necessary experiments, we also hoped to get a sense of the amount of human variability observed across different subjects performing the task, and to compare our quantitative maintainability metrics with the qualitative observations made by the human subjects themselves.

In the remainder of this section, we will present an overview of our experiments and describe our experimental testbed and procedures.

3.3.1 Overview of experiments

As mentioned above, we chose to focus on one maintenance task for our experiments: detecting and repairing a failed disk in a software RAID array. We believe that this is one of the low-level, directly-measurable tasks that would form an axis of the maintainability characterization space as described in Section 3.2.1.

Recall that our primary goal in these experiments is to measure the TIME and LEARNING CURVE costs of this maintenance task. To do this, we applied the procedures described in Section 3.2.3, above. In particular, we carried out a small-scale pilot study using human volunteers. In the study, we first trained the volunteers, then asked them to monitor and repair randomly-injected emulated disk faults. The faults were repeated multiple times, allowing the participant to become more familiar with the task over time, and exposing the learning curve. We monitored the participants' timings and error rates, and collected qualitative observations of their approaches and reactions as well.

Five subjects participated in the study, including one trained system administrator, three computer science graduate students with varying levels of familiarity with the systems, and one professor of computer science. By including a range of subjects, we hoped to be able to observe an example of the extent of human variability when performing maintenance tasks. Note that all of our subjects were members of the ISTORE research group, but none were involved in the design and construction of the experiments.

3.3.2 Experimental setup

As in the availability benchmarks, fault injection played a key role in our experiments. We wanted to present the human subject with a test system that demonstrated realistic failures, but we wanted the failures to be deterministic and controllable for reproducibility and fairness. Furthermore, we wanted to be able to inject failures into any of the disks in a RAID array, not just one particular disk as was done in the availability experiments. Finally, we wanted a system in which we could emulate repair as well as failure, and that could be quickly recovered to a known state after each experimental run.

To meet these goals, we reconfigured the SCSI emulator described in Section 2.2.1 to emulate a set of five separate disks on a shared SCSI bus, and modified its fault-injection capabilities to allow insertion of faults onto any of the five disks. In each experiment, these five disks were ganged together into a RAID-5 array with four data disks and one spare disk. To keep reconstruction time down, the emulated disks were 50MB each, resulting in a 150MB RAID volume. All of the image files were stored on a dedicated NTFS file system at the beginning of a dedicated IBM DMVS18D 18GB 10000RPM Ultra2-LVD SCSI low-profile drive. To support emulated repair, we modified the emulator to simulate the removal of a disk as well as reinsertion of a new disk; the image file backing the emulated "new disk" was always zeroed out before the emulated disk was activated. The emulator was also modified to allow any of the simulated disks to be refreshed from a master snapshot image, allowing us to quickly restore the array to a known and consistent state before each experimental run. Finally, we modified the emulator so that its fault injection and repair functionality could be controlled via a TCP/IP connection over the network; this allowed us to control the timing of the experiment via an external master process that was hidden from the subjects participating in the experiment.

The emulated SCSI disks were connected to the same test system that was used in the availability benchmarks, a PC system with an AMD-K6-2-333 CPU and 64MB of 66MHz ECC DRAM. The emulated disks were attached to an a dedicated Adaptec 2940UW SCSI adaptor. The test system ran one of three operating systems, again the same as in the availability experiments: Windows 2000 Server, Redhat Linux 6.0 with version 0.90-3 of the RAID tools, and Solaris 7 for Intel architectures with Solstice DiskSuite version 4.2. Each system was configured with a five-disk 150MB RAID-5 volume placed

on four of the emulated disks, with the fifth emulated disk serving as a spare. All RAID parameters were set in the same manner as in the availability benchmarks. Each system was also configured with a web server, again as in the availability benchmarks: Solaris and Linux used Apache 1.3.9, whereas Windows used IIS; the web servers were configured with their document root and logs on the RAID volume. The system software for each machine (OS and web server) was installed on a dedicated IBM DMVS18D 18GB 10000RPM Ultra2-LVD SCSI low-profile disk; at any given time, one of the three OS disks was attached to the system via a dedicated Adaptec 2940UW Ultra SCSI controller.

We applied a light web workload to the test system throughout the course of the experiments. This ensured that the various RAID systems were actually using the disks, which is necessary for a disk failure to be detected. The workload that was used was a simple static content workload with one outstanding request and a pause of 200μs between requests. Note that we did not use SPECWeb (as was used in the availability benchmarks) because its file set would not fit on the small RAID volume that we used; furthermore, in these experiments we were not interested in saturating the system or in examining the performance of the workload.

The last part of our experimental setup was a pair of GUI-based control tools, one for the participant in the experiment and the other for the experimenter. Together, these two control tools coordinated the experiment and captured timing results. The first tool was a control interface for the participant, and is shown in Figure 6(a). The main function of this tool was to provide a way for the user to "remove" and "insert" the simulated RAID disks. The tool interfaces with the disk simulator and controls the insertion/removal status of the five simulated disks (numbered 0 to 4, matching the simulated disks' reported logical unit (LUN) numbers). This tool also controls the fault injection sequence used in the experiment (described in Section 3.3.3, below), again by interfacing directly with the disk simulator. It timestamps and logs all injected faults and user insert/remove requests. Also, since it controls the fault injection and the replacement of the disks, the tool can track the "health" of a simulated disk, and thus it can detect when the user accidentally "removes" a healthy drive instead of a sick one; if this occurs, the tool displays an error message and prevents the user from performing the action, but records the occurrence in a log. We
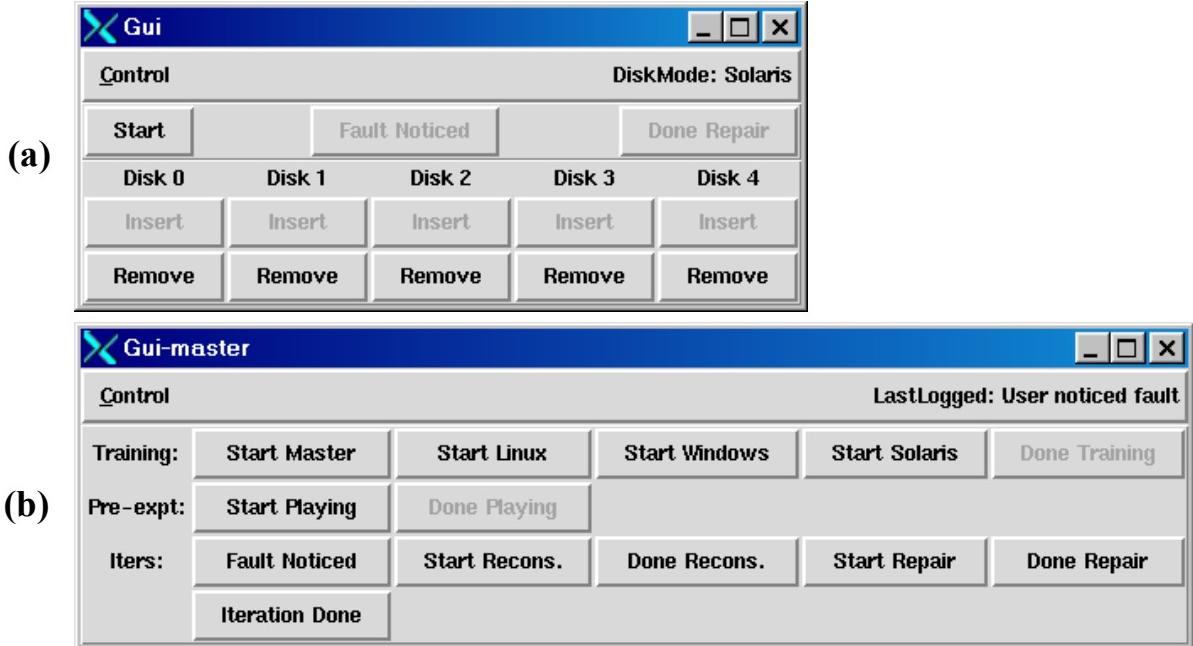
Figure 6: The two control tools used in the maintainability experiments. The upper interface, (a), was used by the subject to insert and remove disks and to initiate the experiment; the lower interface, (b), was used by the experimenter to collect timestamps of important events and user actions.

made this choice because, as was demonstrated in the availability benchmarks, some of our test systems respond badly when two disks fail at the same time.

The second GUI-based control tool was used by the experimenter to record timestamps of the participant's actions (other than those recorded by the participant's control tool). It is shown in Figure 6(b). We originally intended to have the participant record these timestamps via the participant's GUI, but initial experimentation showed that users tended to forget to do this. We therefore decided it would be safer and more accurate to have an external observer recording the timestamps.

3.3.3 Experimental procedure

We followed the same procedure for each volunteer that participated in our experiments. The first step in this procedure was to train the subject. We gave the subject a set of printed training materials formatted as a set of slides, and allowed the subject to study them for as long as they desired. The training material for this initial training step consisted of seven slides that presented an overview of RAID systems, the control GUI, and the general process of replacing a disk in a RAID system. All of the training materials are reproduced in Appendix C.

After the initial training, we generated a random ordering of the three RAID/operating systems, and set up the testbed for the first system. We then gave the subject further training materials, formatted as five slides, describing the system-specific details of the disk-repair task and providing a step-by-step guide for the procedure. These materials are also reproduced in Appendix C. We allowed the subject as much time as needed to digest the system-specific training materials. We allowed the subject to ask questions during the training (both the initial general training and the system-specific training), but not during the actual procedures. The subjects were allowed to write on and to keep the training material during the entire experiment.

After the system-specific training, the subject was introduced to the test system, and allowed to try it out as they desired for an unlimited amount of time. Again, questions were allowed during this "familiarization period." No failures were injected during this period, and the user was not allowed to insert or remove disks.

Once the subject was done with the familiarization period, the actual test procedures began, and lasted for up to 45 minutes. During this time, the following sequence of events was repeated:

1. The system operated normally for a randomly-selected time period of 1–5 minutes.

2. At the end of this period, a fault was injected into a randomly-selected emulated disk.[1]

3. After some further delay, the subject noticed the failed disk, initiated repair, and replaced the disk.

Essentially, the subject was faced with repeated disk failures, and was required to detect and replace the faulty disk each time a failure occurred. The 1–5 minute delay before each failure was designed to ensure that the subject could not predict when a failure would occur, allowing us to test the effectiveness of the failure detection and reporting strategy of each system.

---

1. The type of fault that was injected depended on the system under test, as, due to bugs in our emulator, no single error type was able to cleanly fail a disk across all three systems. Hence, we selected faults that caused the various RAID systems to immediately stop using the faulty disk, resulting in equivalent behavior across all three systems.

At the end of the 45 minute testing period with the first RAID system, the testbed was refreshed to a clean state. Then the entire procedure, starting with the system-specific training, was repeated for each of the two remaining RAID systems.

During the entire experiment, the user was observed by the experimenter (the author), who used the experimenter's GUI to record timestamps of significant events during the test procedures. The entire experiment was also recorded on videotape. The videotaped records contained a running real-time clock that provided a backup source of timing information that proved invaluable for reconciling discrepancies in logged timing data. They also allowed us to capture and analyze the approaches that each user took to maintaining the test systems. A synopsis of the qualitative observations we made from the videotaped records is attached in Appendix B.

We collected a great deal of timing data during the experiments; between the two control GUIs and the videotape, we recorded timestamps defining every phase during each procedure, including system-demarcated phases such as reconstruction onto a hot spare. For our results and analysis, we distilled this data into a measure of *human time* for each iteration of the task, where human time is defined in the same manner as the TIME metric of Section 3.2.2. As discussed in that section, we excluded any waiting times of larger than 1 minute from our human time metric; this 1-minute window was long enough to capture the brief pauses for commands to complete, but short enough to exclude the time for array reconstruction.

3.4 Results and Analysis

In this section we present and analyze our experimental results, focusing on the TIME and LEARNING CURVE metrics. We focus primarily on quantitative results in this section; our qualitative observations of the subjects' behavior are summarized in Appendix B.

3.4.1 TIME

The timing results from our experiments are plotted in Figure 7 for each of the three RAID systems. Each graph shows the human time spent by each subject on each trial of the experiment, where a trial consists of one cycle of disk failure and repair. Certain points are marked with circles; these are trials in which the subject made a fatal error during the
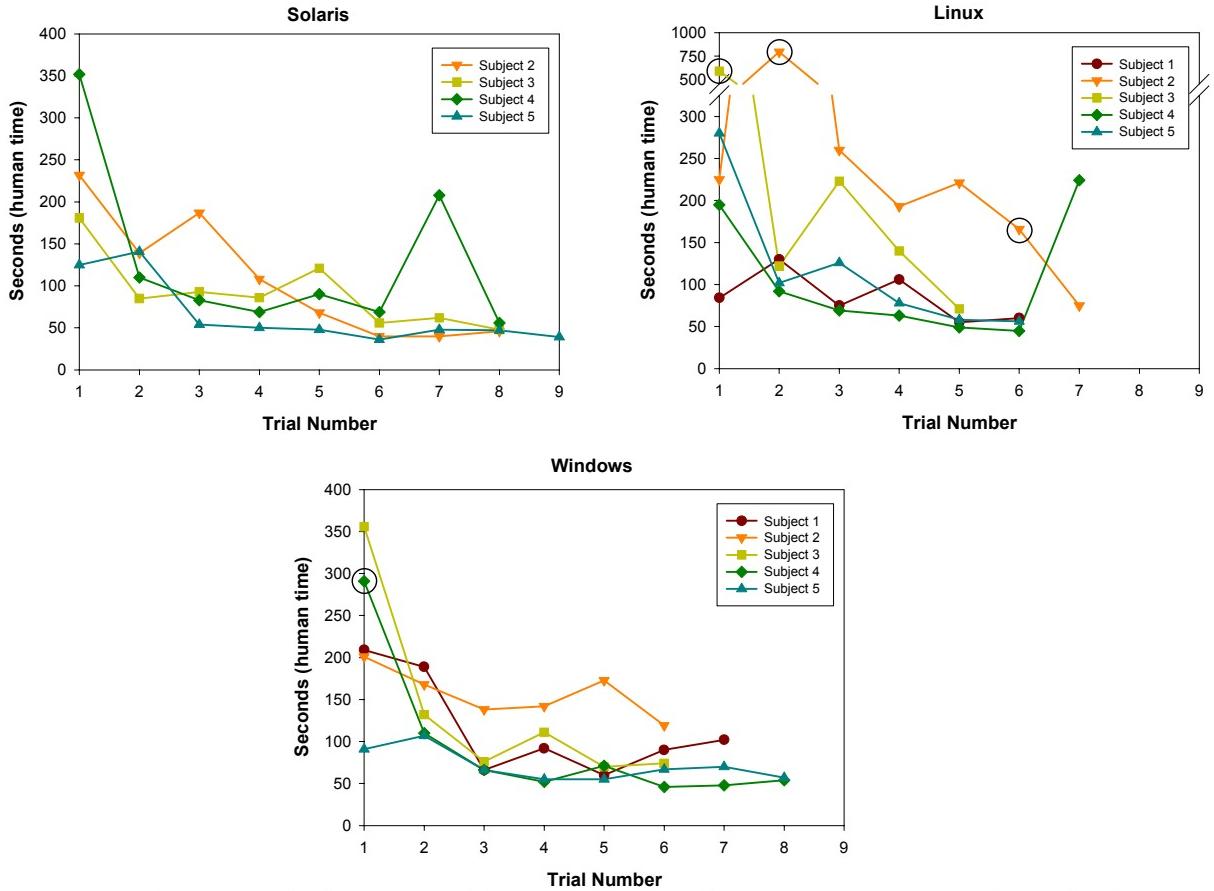
Figure 7: Timing results for maintainability experiments on three operating systems. Each graph shows the human time spent by a set of subjects on each of several trials of the experiment; each trial consisted of one maintenance task: detecting a failed disk in a RAID volume and replacing it. Points marked by circles represent trials in which the subject made a fatal error during the repair procedure. Subject 1 was unable to complete the Solaris experiments due to a failure of the experimental setup.

repair procedure. Note that subject 1 is missing from the Solaris graph due to a failure in the experimental setup; the subject was unavailable for further experimentation once the setup was repaired.

There are several interesting points to observe about these graphs. First, the graphs clearly demonstrate significant variability both between and within subjects, and not only in those iterations where fatal errors were made. However, there appears to be a clear trend toward convergence to a common plateau across subjects as the trials progress, although our sample size and number of trials are a bit too small for the convergence to be complete. The convergence is most notable in the case of Solaris, where by the eighth iteration the times for the participating subjects have a standard deviation of only about 4.5 seconds, less than 10% of the mean. The Windows case shows similar convergence, with very

small standard deviations toward the end, but unfortunately all but two participants had finished before this point so we cannot know whether they would have eventually converged as well. The Linux case does not appear to demonstrate a clear convergence in the statistical sense. However, we believe this is due to particulars of our data set. Subject 4 took an extraordinarily long time on the last iteration due to confusion resulting from an unexpected message from the test system; also, subject 2 appears to be demonstrating convergence but at a much slower rate than the rest of the subjects. These two factors result in a high standard deviation of time for the last few iterations (between 45% and 81% of the mean). If we ignore subject 2 and discard the last data point from subject 4, then the standard deviation of the times across subjects drops to about 15% of the mean in iteration 6, indicating that convergence is likely occurring.

The apparent convergence we observe supports a key assumption that we made in our definition of maintenance cost in Section 3.2.2, namely that there is a system-dependent baseline time cost for each maintenance task to which administrators will converge with sufficient training and familiarity with the system. In fact, this baseline cost is what we defined as the TIME metric, and so the plateaus of the graphs in Figure 7 should be a good indicator of the TIME cost of the disk repair maintenance task.

Ideally, we would like to extract and compare numerical estimates of the TIME costs of the disk repair task across the three systems. To do this, we apply the human-experiment approach identified in Section 3.2.3 for measuring TIME, and calculate a plateau value for each graph based on the minimum time demonstrated by the human subjects.

Our plateau-finding procedure works as follows. For a given operating system platform, we compute the minimum time spent by each subject on the task across all iterations. In our case, this gave us either four or five values for each of three operating systems, depending on the number of subjects that participated. We then averaged those values for each operating system, giving us a TIME score for each system. Note that we use an average here (rather than taking the minimum of the minimums) in order to preserve and analyze the effects of human variability across subjects. The average scores are listed in Figure 8. The table also includes standard deviations and 95% confidence intervals computed under the assumption that the minimum time values across subjects are normally distrib-

Metric, in seconds	Solaris	Linux	Windows
Mean TIME score	45.0	60.4	70.0
Standard deviation	8.9	12.4	28.7
95% confidence interval	45.0 ± 12.3	60.4 ± 14.2	70.0 ± 33.0

Figure 8: Values of the TIME metric computed for each of the three test systems. The TIME score is computed as the mean of the minimum times turned in by each subject on the maintenance task of detecting and replacing a failed disk in the RAID array. All values are reported in seconds. The 95% confidence interval describes the range of values in which the mean TIME score is expected to fall 95% of the time, were the experiment to be repeated a large number of times. Notice that the standard deviations are quite large, particularly for Windows, and that all three confidence intervals overlap, implying that we cannot conclude a statistically-significant difference between the systems.

uted. Note that, due to our small sample size, we have no good way to test the normality assumption.

The first thing to note from the results in Figure 8 is that, according to the means, Solaris has the best TIME score, followed by Linux and then Windows. This would imply that, on the task of detecting and repairing a failed disk in a RAID array, Solaris has the least time cost for the administrator, and thus the highest maintainability marks along that axis. Unfortunately, this conclusion cannot be statistically supported with the small number of subjects present in our experiments, as can be seen by the overlapping confidence intervals across the three TIME scores.

We have carried out a more detailed statistical analysis of the data in order to get a better handle on the TIME differences between the systems. Our analysis is based on statistical hypothesis testing. We start with the hypothesis that Solaris has a statistically better TIME score than Linux. That is, the difference between Linux's average TIME score and that of Solaris is statistically greater than zero. To test this hypothesis, we examine the inverse null hypothesis that the difference between Linux's and Solaris's TIME scores is statistically less than or equal to zero. Symbolically, if t_S represents Solaris's score and t_L represents Linux's, then our null hypothesis is H_0: $t_S \geq t_L$. Note that we want to *reject* this hypothesis: if we do so, then we know that in fact $t_S < t_L$, which is what we want to show. The reason for this seemingly-backwards approach is that it is much easier to statistically reject a hypothesis than it is to statistically prove one.

To examine H_0, we carry out a statistical procedure known as a paired-t analysis; we summarize the results here, with the details of the procedure and the analysis left to Appendix A. In carrying out the analysis, we found that the test of the null hypothesis

H_0: $t_S \geq t_L$ had a *P-value* of about 0.093. The *P*-value is a measure of the confidence of test's result: what the *P*-value of 0.093 means in this case is that there is a 9.3% chance that we would observe the data in Figure 7 if the null hypothesis is *true*, that is, that Linux has a *lower* time cost than Solaris (the opposite of what we want to prove). What this boils down to is that we can *reject* the null hypothesis H_0 at only a 90.7% confidence level, less than the typically-used standard of 95% confidence (recall that by rejecting H_0, we implicitly confirm that $t_S < t_L$, which is what we want to show).

Mapping this all back to more meaningful quantities, what this 90.7% confidence level means is that, if indeed Solaris intrinsically has a lower TIME cost than Linux, and if we were to repeat our experiment many, many times (with a different, randomly selected set of 4 subjects in each case), then we would expect to see that about 90.7% of those experiments would actually produce results showing that Solaris has a lower TIME cost than Linux. This is a somewhat low percentage—it means that about 9.3 out of 100 experiments will generate results that wrongly show Linux to have a lower TIME cost than Solaris. Typically, we like to have no more than a 5% probability of error—no more than 5 of 100 experiments giving the wrong result—before we accept the result of any one experiment. Thus based on the data from our single experiment, we cannot conclude at the standard 95% confidence level that Solaris truly has a lower TIME cost than Linux, although we could do so at the less rigorous 90% confidence level.

The reason that our conclusion is weak is due to the small number of data points—in this case, only four, since one subject did not complete the Solaris experiments. We can, however, compute how many subjects we would expect to need if we wanted to get a statistically conclusive result (at the 95% confidence level) that Solaris's TIME cost is lower than Linux's. To do this, we carry out a power analysis. The basic idea of a power analysis is to compute the sample size (number of subjects) needed in order to reject a hypothesis with 95% confidence. The details of our analysis are again deferred to Appendix A, but the results state that, in the case of the comparison between Linux and Solaris, we would need $n \geq 6$ in order to reject the null hypothesis H_0: $t_S \geq t_L$ at the 95% confidence level. So 6 subjects, only 2 more than we actually used, would be sufficient to get a statistically sound conclusion.

Claim	Supported at 95% confidence?	P-value	Subjects needed for 95% confidence
Solaris < Linux	No	0.093	6
Linux < Windows	No	0.165	14
Solaris < Windows	No	0.118	7

Figure 9: Summary of statistical analysis of the TIME **metric.** The first column lists possible claims we would like to make; "*x* < *y*" means that we would like to claim that *x*'s TIME metric is less than *y*'s value in a statistically-significant sense. None of the three claims is supported at the standard 95% confidence level. The *P*-values are taken from a paired-*t* test of the contrary hypotheses, namely that "*x* ≥ *y*"; *P*-values of less than 0.05 mean that the contrary hypothesis is rejected, and thus the claim is supported. The last column lists the expected number of experimental subjects that would be needed to support the claim at 95% confidence, assuming the measured means and standard deviations.

So at this point we have thoroughly analyzed the Solaris/Linux comparison case and have concluded that, although there is suggestive evidence that Solaris's TIME cost is less than Linux's, we would need a slightly larger experiment to have statistical certainty. We carried out a similar analysis for the other pairings of the systems; the details of the analysis are similar and are therefore omitted. What we found was the following. In comparing Linux and Windows, the straight averages suggest that Linux's TIME cost is lower than Windows's. However, the *P*-value computed to test the contrary hypothesis was only 0.165, meaning that we can make that conclusion only at a confidence level of about 83%, much lower than the standard 95% level. Using a similar power analysis to that introduced above, we found that we would need approximately 14 subjects to bring the confidence level up to 95%. Finally, we directly compared Solaris and Windows; again the averages in Figure 8 suggest that Solaris's TIME cost should be lower than Windows's, but the statistics do not support that conclusion at the 95% confidence level. In this case, the *P*-value computed to test the contrary hypothesis was 0.118. Applying a power analysis to the test suggests that an expanded experiment with 7 subjects would be sufficient to just push the test over the 95% confidence level.

The results of all of the statistical analysis of the TIME metric are tabulated in Figure 9. In summary, if we are willing to drop our confidence level to about 83%, then we can conclude that there is a statistically significant difference between the TIME costs for performing the disk repair maintenance tasks across the three systems, and in particular Solaris has the best (lowest) time, followed by Linux and then Windows. We would need similar results from a larger experiment with 14 or more subjects in order to draw these

Error Type	Windows	Solaris	Linux
Fatal Data Loss	💣		💣💣
Unsuccessful Repair			💣
System ignored potentially-fatal user input			💣
User Error — Observer Intervention Required	💣	💣💣	💣
User Error — Subject Recovered On Own	💣	💣💣💣💣	💣💣
Large Software Anomaly			💣💣
Small Software Anomaly		💣	
Total number of trials	35	33	31

Figure 10: Errors made by subjects during benchmark trials. Each 💣 symbol represents a single occurrence of an error. Errors are aggregated across subjects. There are 9 types of error. In order, the first two error types are fatal errors that would result in data loss. The third error type represents cases where the subject's actions should have resulted in fatal data loss, but did not due to inexplicable behavior of the test system. The two "User Error" types represent cases where the subject got confused or performed an incorrect but non-fatal action; we break these down by whether or not the external observer had to intervene to get the subject back on track. The last two error types represent cases where the RAID system software behaved differently than expected due to an unexpected sequence of user input.

conclusions at the standard 95% confidence level. But, given the data we have, it is suggestive that the TIME metric is capable of distinguishing a difference in maintainability across the three systems.

3.4.2 LEARNING CURVE

Now that we have quantified and compared the TIME metric across the three systems, we turn to the other metric of interest: LEARNING CURVE. We will consider two of the three factors that we defined as part of LEARNING CURVE in Section 3.2.2. The first is the time it takes for the subjects to converge to the optimal time (the plateaus discussed in the context of the TIME metric, above). From visual inspection of the graphs in Figure 7, it appears as if Solaris and Windows both converge significantly faster than Linux, with Solaris having perhaps a slight edge. Unfortunately, due to the large variability between subjects and the small size of our sample, we cannot draw a more rigorous conclusion about the convergence time.

The second factor of LEARNING CURVE that we will consider is the number of errors made by the subjects as they repeatedly performed the disk-repair maintenance task. We start by looking at an overall distribution of errors, aggregated across subjects and iterations, but broken down by error type. This data is represented in tabular form in Figure 10. Looking first at the top two rows of the chart, which represent fatal errors

where data would be lost, we see that Solaris does best: no subject would have caused the system to lose data on any of the 33 experimental trials of Solaris. Windows comes second, with only one fatal error out of 35 trials, whereas Linux is significantly worse, with three fatal errors out of its 31 trials. It is interesting to note that the one fatal error for Windows came in the first trial, whereas Linux's fatal errors came in the first, second, and sixth iterations. The fact that subjects were still causing fatal errors in Linux as late as the sixth iteration suggests that the learning curve for Linux is relatively flat, and that the errors are due to inherent maintainability complexity in the system, rather than to unfamiliarity of the users. We will return to this important point further, below.

Now let us consider the entirety of the data tabulated in Figure 10, including the non-fatal errors. If we look at this overall error distribution, we see that in this case, Windows does the best, with only three errors total out of 35 trials. Solaris is next, with 7 errors out of 33 trials, and Linux again comes in last with 9 errors out of 31 trials. Comparing Windows and Solaris, we might hypothesize that Windows's maintenance interface for the disk-repair task does a better job of guiding the user through the task, resulting in fewer overall errors, whereas Solaris's interface, while more error-prone than Windows's, offers better protection against fatal errors.

This hypothesis is in fact borne out by our qualitative observations of the scenarios resulting in each of the errors. In the Windows case, the fatal error was due to the subject removing the wrong disk at repair time; according to the subject, this happened because Windows numbered the on-screen representation of the array disks using a 1-based sequence, whereas the drives were physically numbered in a 0-based sequence. Because Windows obscures the mapping between a visual representation of the disk on-screen and the actual device number (LUN) of the disk in the array, it introduces the possibility of fatal error during the repair procedure. Also, Windows also presents a GUI-based disk maintenance interface that constrains the user to a certain sequence of repair steps; our subjects confirmed that it was difficult to perform the task incorrectly using this interface. In contrast, Solaris uses a command-line interface that makes it possible to accidentally skip or reorder steps, leading to a greater number of overall errors. But, Solaris incorporates the array disks' physical numbering scheme directly into its logical drive names, virtually eliminating the chance of a fatal error like that made on the Windows system. If we
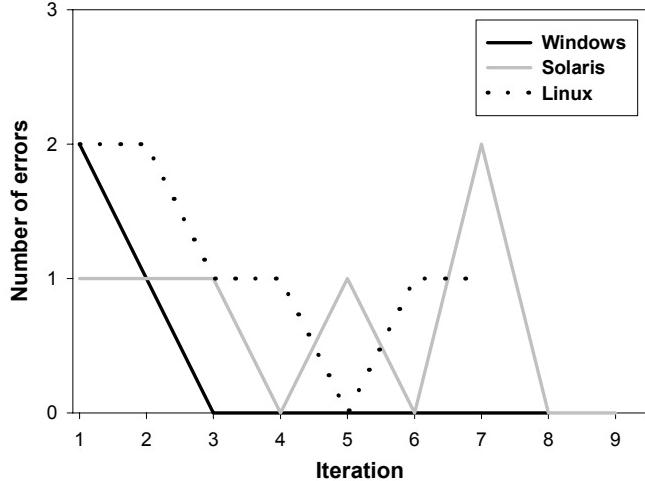
Figure 11: Time series view of user errors. The graph plots the number of errors (fatal and non-fatal) made by subjects attempting to perform the disk-repair maintenance task on each of three RAID systems. System anomalies were excluded from the data. Error counts for each iteration (experimental trial) were aggregated across subjects. Notice that Windows is the only system that demonstrates the expected "learning curve" behavior of a quickly-falling error rate with no errors in the later iterations.

bring Linux into the discussion, we see that it has both a high overall failure rate and a large number of fatal errors. Again drawing on anecdotal reports from the subjects themselves, we find that this is due to the fact that Linux has both a complex, error-prone command-line interface like Solaris, as well as a complex disk naming scheme that buries the actual physical disk number under *two* layers of indirection.

Finally, we look at the error data in time-series form in order to see the true learning curve effect, and to answer the question of whether error count decreases with increased familiarity with the system. Figure 11 plots the number of user errors (excluding system anomalies) in each trial for each RAID system, aggregated over all subjects. Some very interesting trends are immediately obvious. First, the curve for Windows follows exactly what we would expect for a learning curve: the errors are concentrated in the early iterations, and fall off to zero after the second iteration. This confirms part of the hypothesis made above, that Windows's maintenance interface is well-designed and easily learnable. It also implies that there is not a lot of inherent complexity in the maintenance procedure or interface under Windows, since all of our subjects quickly mastered the task, and remained error-free throughout the later trials of the experiment.

If we look at the behavior of Solaris and Linux, however, we see a different story unfolding. Neither system exhibits a learning curve of the expected shape; in fact, our subjects surprisingly continued to make errors with both systems throughout the set of exper-

imental trials. This suggests two points: first, that the disk-repair maintenance interfaces on these two systems are not as well designed as Windows's, and, more importantly, that these systems suffer from greater maintenance complexity. This complexity point is supported by the prevalence of errors late in the set of experimental trials, which indicate that the maintenance procedures and interfaces on Solaris and Linux are too complex for the subjects to master, causing them to continue to make errors despite a long familiarization period with the systems.

Summarizing our analysis of the LEARNING CURVE data, we can conclude that the Windows 2000 system is the clear winner in terms of maintainability on the LEARNING CURVE metric. It was the only system that showed a decrease in errors over time, indicating a low inherent complexity of the maintenance task and a well-designed maintenance interface. The one weak point of Windows is that its interface obscures the mapping of logical disk "entities" in the administrative interface to the actual disks in the system, raising the potential for fatal data loss errors. In contrast, Solaris avoids this mapping obfuscation, and as a result is the only system to not suffer any fatal data loss errors. However, Solaris does have many more non-fatal errors and an essentially-flat learning curve, demonstrating inherent complexity in its maintenance task procedure and maintenance interface. Finally, Linux behaves the worst of the three systems, suffering from both a bad case of disk-name obfuscation and a high-complexity interface and maintenance procedure. These flaws manifest themselves as high rates of both fatal and non-fatal errors.

3.4.3 Discussion

Summarizing the results of our experiments and analysis, we find that, on the TIME metric, Solaris has the lowest maintenance cost, followed by Linux and then Windows. Along the orthogonal axis of the LEARNING CURVE metric, the ordering switches: Windows now does best, followed by Solaris and then Linux. Amazingly, despite being based on a simple analysis of time and error rates, these results confirm the qualitative observations made by our subjects during the experiments, which are detailed in Appendix B. Looking first at the TIME rankings, the comments made by our subjects during the experiments suggest that they felt that Solaris was the most efficient system to use, due to its well-designed command-line interface. This efficiency came from a number of sources, notably the

scriptability of the interface, the directness of the disk-naming convention, and the clarity of the RAID status displays. While Linux also used a scriptable command-line interface, more than one subject felt that its commands, procedures, and status output were confusing and unclear, reducing efficiency somewhat compared to Solaris. The subjects's feelings on Windows's efficiency also confirm its order in the rankings: while most subjects praised its simplicity, several noted that it was not scriptable, leading to a more manually-intensive repair procedure.

Our experimental conclusions from the LEARNING CURVE metric are also supported by observations made by our subjects. Although two of the subjects complained that Windows was not scriptable, all but the trained UNIX system administrator agreed that the Windows interface was the simplest to understand and master. Several subjects noted that Windows guides the operator through the maintenance process by restricting the number of choices at each step of the procedure, making it harder to commit a fatal error. These observations support Windows's first-place ranking in our LEARNING CURVE analysis, and illustrate why the system exhibited the expected learning-curve behavior of quickly-decreasing error rates. In contrast, both command-line systems suffered from a lack of guidance: our subjects agreed that neither Solaris nor Linux provided feedback as to the legal operations at each step of the repair procedure, resulting in a greater number of user errors, even in the later iterations of the experiment. Linux did worse than Solaris, because, according to our subjects, its interface suffered from a more complex disk naming scheme, less-clear status displays, and more complicated procedures. The poor fate of the command-line systems on our LEARNING CURVE evaluation scale is an important result, confirming that the inherent complexity of a system's interface can result in a learning curve that never approaches an error-free plateau.

Looking at all of our results, we can infer some guidelines for increasing the maintainability of a repair task, at least in terms of the TIME and LEARNING CURVE metrics. To decrease the TIME metric, a system should above all provide an interface that is exceptionally clear, with no extraneous detail presented, and with logical views of the system that can be quickly and easily mapped to the physical components. The interface should be scriptable to reduce interaction time. To do well on the LEARNING CURVE axis, a system should provide guidance at every step of the repair procedure. For GUI-based systems,

this can be done by modifying menus and screen displays to only show the legal steps at each point in the repair procedure. Command-line systems could achieve similar results by displaying the set of legal commands at the end of each step's execution, or by combining the commands into unified scripts that each handle an entire repair task. Finally, every effort should be taken to reduce the complexity of the task, by hiding extraneous detail not needed in the common case, by providing clear status information, and by naming devices and objects in ways that map directly to their physical identifiers.

All of these suggestions merely act as a way of papering over the problems with current system designs, however. Perhaps it is also time to consider more radical changes in the way we build systems, focusing on incorporating maintainability-enhancing features from the ground up. For example, for all of the RAID systems we examined, it was much too easy for both novice and experienced users to cause fatal data losses. It was also trivial for users to make non-fatal mistakes in the repair procedure from which it was difficult to recover. Most of these errors could have been mitigated if the systems offered a way for maintenance actions to be undoable up until an explicit commit point, much in the same way that most word processors allow undoing edits up until the active file is explicitly saved to disk. In such a system, rashly-executed commands and erroneous inputs would not take effect immediately, and the administrator would have a chance to review the captured set of actions before it was actually carried out. Furthermore, the system itself could check the entirety of the captured set of actions for consistency, flagging for administrative review any sequences of actions that would result in a loss of data or availability.

Another radical way to improve maintainability would be to include systemic support for simulated maintenance. One of the reasons why maintenance (especially emergency maintenance) is expensive and dangerous is that it requires human beings to perform unfamiliar tasks in high-pressure situations. Imagine designing a system that had something like our maintainability benchmarks built-in: when placed into a special *training mode*, the system would simulate faults and require the user to carry out a realistic repair procedure. With safeguards added to prevent damage from errors made during the repairs, such procedures could go a long way towards improving administrator training and familiarity with key repair procedures, as well as identifying problematic procedures that need

redesign. Systemic support for this kind of simulated maintenance would also greatly simplify maintainability benchmarking, making it more likely to be used in practice.

Finally, we can imagine building systems that simply require less error-prone human maintenance. Clearly, as systems are more able to maintain themselves, the cost of human maintenance will decrease and the rate of maintenance-induced errors will drop. Certainly, maintainability benchmarking will be easier as fewer tasks will exist that require human experiments in order to measure them. There are a myriad of possible ways to improve system maintainability, running the gamut from self-diagnosis techniques like automatic root-cause analysis to self-repair techniques that use spare resources to fix discovered problems. Self-maintaining systems are a wide-open area of future work, with many ideas yet to be discovered and potentially great rewards for those who find them.

3.5 Future Directions

In this final section, we discuss the extensions needed to take our maintainability characterization methodology to a full application-specific maintainability benchmark, then stop to reflect on the practicality of the approaches we have defined throughout this report.

3.5.1 Extensions to the methodology

Recall that a full application-specific benchmarking methodology has three main components: characterization of the system, characterization of the workload, and distillation of those characterizations into a single metric. So far, we have only discussed a simplified version of the first step, system characterization when the axes of characterization (the system's possible maintenance tasks) are known. Thus there are three more challenges to solve: enumerating all possible maintenance tasks, characterizing the workload by measuring the frequency of each maintenance task for a particular system/site installation, and defining an overall cost function that combines the measured task maintainability costs with the task frequencies and reduces them to a single maintainability metric. While we have not implemented or rigorously defined these remaining pieces of the methodology, we offer some of our initial thoughts and ideas.
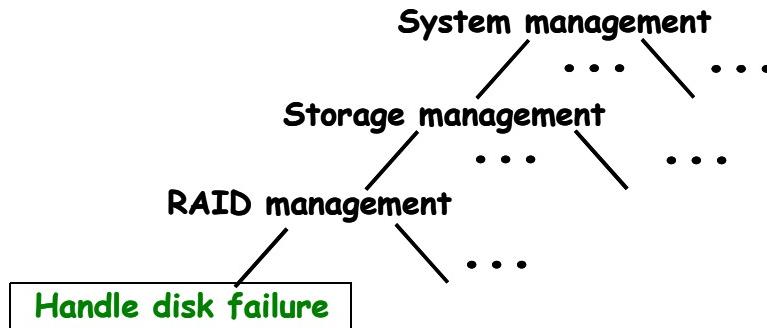
Figure 12: A slice of a possible taxonomy for maintenance tasks, showing the location of the particular low-level maintenance task measured in our maintainability experiments.

Starting with the problem of enumerating all possible maintenance tasks, we are faced immediately with two challenges: the sheer size of the space of possible tasks, and the fact that only a small fraction of that space applies to any particular system. We believe that the best way to deal with the size and sparseness is to consider a hierarchical taxonomy of maintenance tasks with actual measurable tasks at the lowest level of the hierarchy. For example, our task of repairing a failed disk in a RAID array would appear in a slice of the taxonomy that might look like what is depicted in Figure 12. For systems that did not require storage management, that entire part of the taxonomy could be ignored.

Although it sounds daunting to construct this task taxonomy, two factors work to our advantage. First, the taxonomy can be built incrementally as needed; indeed, in this study we were able to carry out a useful and revealing maintainability benchmark with only the coarse taxonomy depicted in Figure 12.[2] Second, there is an existing body of work that provides a good basis for construction of the taxonomy. As part of their efforts on system-administrator certification, the SAGE Systems Administrator's Guild has carried out a survey that breaks down a system administrator's responsibilities into 14 very broad categories, such as "network maintenance" and "database administration" [37]. These categories can serve as the very top level of the task hierarchy. At a more detailed level, Anderson and Patterson have extracted a two-level task hierarchy by analyzing and categorizing the papers that have appeared in the past 12 years of proceedings from LISA, the leading systems administration conference [2]. Their hierarchy divides tasks into 64 categories such as "service administration: printing" and "configuration management: host configura-

---

2. Note that, while a coarse taxonomy like this one is adequate when the goal of the benchmark is to understand a specific part of the maintenance problem or to compare a specific task across different systems, the full taxonomy will still be needed if the goal of the benchmark is to compare the overall maintainability of several different systems across all tasks.

tion". While still relatively broad, these categories have enough definition that it should be possible to enumerate the specific tasks contained within them, thereby completing the lowest levels of the task taxonomy.

The next problem we must address is how to characterize the workload by measure task frequencies for a particular system installation. This is an important step in the methodology; frequency weightings provide an indication of which tasks are most important, thereby providing a means of evaluating if an expensive task is indeed a significant contributor to the system's overall maintainability cost. Unlike the system characterization that we have focused on up to now, the frequency measurements depend on both the system and the environment (or site) in which it is installed.

Task frequencies can be measured in a variety of ways. The simplest is via surveys: one can interview administrators and find out what maintenance tasks are done at what frequency. This approach has been used reasonably successfully in the past [1] [15] [28], although most existing surveys have collected data at a coarser granularity than what we would need to get frequencies for the bottom level of the task hierarchy.

Besides the granularity issues, surveys also suffer from implementation problems. Most notably, they are complex and slow to administer, and survey respondents often misrepresent data that reflects negatively on them. Thus another approach that can be taken is to use logs from the system to determine how often administrative tasks were performed [34]. This approach has the advantage of not requiring subjective recall by the administrator, but suffers from the disadvantage that logged information is often not complete enough to allow identification of all maintenance tasks, particularly when the tasks were performed to resurrect the system from some catastrophic failure state.

Ideally, some combination of surveys, log analysis, and third-party monitoring of installations should be used to gather task frequency data. As this is clearly an expensive and time-consuming chore to carry out for every maintainability benchmark experiment, it may make sense to define some fixed task frequency profiles that are reasonably representative of certain classes of systems and installations, much as fixed workload profiles are used today in performance benchmarks. Further investigation will be required to determine how representative and effective this approach might be.

Finally, we address the issue of what to do with all of the data that results from the system characterization and site frequency profile. In an application-specific benchmarking framework, the goal is to combine and reduce that data into a single (or small set) of metrics that represent the system's overall maintainability when deployed in the profiled site.

The simplest way to accomplish this is to simply take a weighted average of the measured task costs, using the measured frequencies as weights. This approach makes sense for the TIME and IMPACT task cost metrics, but unfortunately does not apply directly to LEARNING CURVE. However, there may be a way to incorporate the LEARNING CURVE data along with the other metrics. Recall that TIME and IMPACT measure *minimum* costs, assuming perfect human behavior; the LEARNING CURVE has the data that tells us how far from perfection we expect a human administrator to be. In particular, we can use task frequency to index the learning curve, thereby obtaining a weighting factor that can be applied to the baseline TIME and IMPACT metrics. The intuition is as follows: frequently-occurring tasks will likely be carried out by administrators working efficiently at the end of the learning curve, while rare tasks fall toward the start of the learning curve, where administrators are likely to work more slowly and be more error-prone. By using the measured time dilation and error probability at the frequency-indexed point on the learning curve, we can adjust the baseline TIME and IMPACT scores to reflect realistic estimates of human behavior; these adjusted scores can then be combined in a weighted average as described above for the raw scores.

Of course, all that we have proposed in this discussion is speculative, as we have not carried out any experimental verification of these extensions to our basic task characterization methodology. Hopefully these proposed extensions will provide plentiful fodder for future investigations into maintainability benchmarks, and will lead to a demonstration of a complete maintainability benchmarking methodology.

3.5.2 Towards a more practical maintainability benchmark

One of the great challenges in building maintainability benchmarks is simply the cost and difficulty of carrying them out. Just the experiments described in Section 3.3 for a *single task* on the software RAID systems required five human subjects and about one to two man-weeks worth of time to reconfigure the existing testbed, design and test the experi-

ments, and carry them out with the subjects. If we had to build the testbed from scratch, the time would have been significantly longer. Recall also that we would have needed at least 14 randomly-chosen subjects to get a statistically-significant conclusion on the TIME metric, further lengthening the amount of work and time.

While this amount of work is probably acceptable for measuring the maintainability of a single, important task of interest, it makes it unlikely that we will ever use this methodology for benchmarking overall system maintainability in the same way that we benchmark performance. If we were to carry out a full maintainability benchmark along the lines of the extensions we just described in the previous section, we would have to enumerate every maintenance task appropriate to the system, and then carry out experiments like those of Section 3.3 for each of those tasks. This is a staggeringly ambitious proposal.

And yet, the very aspect that makes this methodology expensive is also what makes it work: the use of human experiments to quantify the human effects (such as LEARNING CURVE) on maintainability. In our RAID case study, for example, it is unlikely that we would have been able to arrive at our LEARNING CURVE conclusions without the human experiments. While it may be possible to approach some of the human issues analytically, for example by measuring the branching complexity of the paths through an interface, such an analysis will never capture the full extent of human variability and behavior.

Another drawback to our methodology is that it requires that maintenance tasks be identifiable and reproducible. While a large fraction of maintenance tasks do fall into this category, some important ones do not. In particular, system administrators can spend a great deal of time attempting to diagnose the low-level root cause of a problem based on some observed high-level symptoms. These diagnosis tasks are not captured easily in our model of concrete maintenance tasks, and are not easily measured in an iterative fashion. Although we can most likely generate diagnosable problems via fault-injection, further work will be needed to identify appropriate sets of problems and the corresponding fault injection needed to create them. Also, it is also not clear how we would train inexperienced human subjects to be able to carry out the diagnosis—it may be necessary to use only trained administrators for diagnosis experiments, further complicating the experimental process.

There is a potential solution to all of these difficulties with our methodology, and it offers an interesting direction for future work. The basic difficulty with our methodology is that it tries to measure all of maintainability in a system-specific way. If we look at benchmarks in other areas, such as performance, we see that they have traditionally addressed similar problems by defining away some of the scope: even the most-respected performance benchmarks like the TPC suite measure only a small component of the entire performance space, and do so with a static workload representative of only a single environment. While this limiting of scope has led to a lot of debate about the validity of performance benchmarks, and to proposals to restore environment/application-specificity to benchmarks [42], it has clearly simplified the benchmarking process and allowed performance benchmarks to become pervasive.

As long as humans are involved in the experiments, we will never be able to achieve the same level of ubiquity with maintainability benchmarks as we see with performance benchmarks. However, we can get significantly closer if we borrow some of the ideas of narrowing the scope of the methodology. One interesting proposal on how to do this was recently raised by James Hamilton of Microsoft [22]. Hamilton observed that some popular performance benchmarks, notably the TPC benchmarks [49], already require a great deal of expense, time, and maintenance to carry out. However, because the results of these benchmarks are so important, companies are willing to go to extraordinary expense to carry them out, typically building system-specific software and infrastructure just for the benchmarks. Hamilton therefore proposed adding an optional "maintainability test" to these benchmarks, in which fault-injection-based maintainability experiments are carried out as a second pass after the main performance data has been collected. Because benchmarkers are already building system-specific testbeds and software for the performance aspects of these benchmarks, the marginal cost of adding maintainability-benchmarking features (like fault-injection) to the testbed may be low. Also, the TPC benchmarks, while reasonably narrow in scope, are well-accepted by the community as being representative enough to be useful.

The last problem is handling the human aspect of maintainability experiments; the solution here is to have each benchmarker make available a fixed number of their best system administrators during the maintainability benchmarking runs. While the use of best

administrators may not capture the full variability amongst all possible administrators, it should capture a lot of it, especially the variability demonstrated within each administrator. Furthermore, the notion of "best administrators" is actually well-defined in this case, as benchmarks like TPC-C are typically carried out by the companies who design and sell the systems being benchmarked; such companies have access to well-trained, highly-skilled administrators and can (and do) supply them for the benchmarks.

While Hamilton's approach cannot bring maintainability benchmarking to the ubiquity of popular performance benchmarks like SPEC or *lmbench* that can be run by anyone, it certainly provides a more palatable initial path towards popularizing the important idea of maintainability benchmarks, and provides a way for the benchmarks to gain acceptance in the arena of large-scale servers, where maintainability is especially critical.

Section 4

Related Work

4.1 Availability Benchmarks

The notion of benchmarks to measure system availability is a relatively new one to the system community. However, with the community's recent shift in focus toward issues of reliability and robustness, system evaluations that include rudimentary availability testing are beginning to be published. For example, Saito's experimental evaluation of the Porcupine e-mail system includes a "failure recovery" analysis [38]. The Porcupine analysis looks at the performance impact of coarse-grained fault injection (node failure and recovery) using a graphical representation that is very similar to the quality-of-service graphs that we have proposed. Gribble's analysis of a distributed data structure implementation (published after this work's initial publication) uses the same technique and graphical representation [21]. Although both the Saito and Gribble work share the same representation of results with this work, and can also be considered availability benchmarks, our work differs in that it attempts to define a general methodology for availability benchmarking. Our work also takes a more system-wide, black-box approach, applying faults based on observed real-world fault distributions to elucidate possible availability problems with a system, rather than being targeted to test a particular known recovery mode of the system.

Although the notion of benchmarks to measure system availability or "robustness" may be new to the systems community, it has not been neglected by the fault-tolerance community. Siewiorek describes "robustness benchmarks" based on fault injection per-

formed primarily by using an application to feed corrupt input to the system [44]. Tsai, working on Tandem machines, proposes another set of reliability benchmarks based on software-implemented fault injection and a synthetic workload generator. His metrics include an average measure of performance degradation due to faults, a simpler precursor of our time-dependent quality of service metrics [50]. Koopman describes benchmarks to test OS robustness by feeding corrupt data to system calls [29]; similarly, Miller et al. and Forrester describe the Fuzz tests, robustness tests based on feeding random data or messages ("fuzz") into key input points in operating systems and applications [17] [32] [33].

The major difference between these benchmarks and the ones we propose is in their goals and the knowledge they assume. Tsai's and Siewiorek's benchmarks are primarily designed to test particular known fault-tolerance mechanisms deployed in fault-tolerant hardware and software systems; to this end, their benchmarks target and evaluate specific components, layers, or mechanisms in the system under test, and thus assume knowledge about the error-detection mechanisms and general structure of that system. In contrast, our benchmarks take a more black-box approach, assuming little about the system under test (not even that it is fault tolerant), and applying faults designed to match real-world failure patterns. Koopman's benchmarks and the Fuzz tests do this as well, but are limited to faults generated by passing random, corrupt data to system calls and input routines; we try to mimic more general faults, including realistic hardware failures.

An additional key difference is that our availability benchmarks measure the system's availability behavior in terms of application-specific metrics that reflect quality of service visible from the client's point of view. Finally, our multi-fault workloads go beyond the isolated faults examined by Siewiorek, Tsai, and Koopman by relating the behavior of a system to realistic scenarios that affect large-scale server systems and by providing a foundation for the expansion of the benchmarks to incorporate the measurement of maintainability.

The techniques of fault injection that we use are also not uncommon in the fault-tolerance community, where fault injection is commonly used in a case-specific manner to verify fault tolerant systems, to generate models of fault tolerance behavior, and to study fault propagation [4] [11] [12] [14] [27] [35]. However, most of this work uses either very low-level hardware fault injection that requires expensive and dangerous equipment (such

as heavy-ion bombarders) [11], or software-implemented fault injection. The former is not tractable for general use because of the cost and complexity, and the latter is not particularly portable, as it generally requires modifications to the OS or driver layer. In contrast, our approach of hardware fault injection at standard interfaces (such as the SCSI-level fault injection used for the RAID study) is both portable and relatively simple; for example, we could have easily used the same fault-injection setup (consisting of off-the-shelf PC hardware and software) to measure the availability of software RAID on a SPARC/Solaris machine.

Finally, before the time that this work was initially published, there had been several studies of RAID reliability and availability [25] [26], but these had focused on simulation studies of hardware RAID, and none had examined RAID in the context of a general availability benchmark. Following on this work's initial publication [10], researchers at EMC, a major commercial RAID system vendor, adapted our availability benchmarking techniques to measure the availability of the centralized cache component in an undisclosed EMC storage system, and found the techniques useful in identifying availability weaknesses as well as pointing out flaws in the system's diagnostic tools [31]. We applaud EMC's willingness to carry out these benchmarks and to publish the results, and we hope that this is the first step toward gaining commercial acceptance for availability benchmarks.

4.2 Maintainability Benchmarks

The area of benchmarking maintainability is a very new one, and thus there is very little existing work that focuses directly on the subject. There is one notable exception, though. Along with the availability study mentioned above, EMC has carried out some initial maintainability benchmarks on the cache component of one of their storage arrays [31]. In their study, the researchers injected faults into the system, then attempted to diagnose and repair any resulting damage themselves, following the documented procedures used by field service engineers. Afterwards, maintainability scores were calculated for the simplicity of the repair task, on the effectiveness of the diagnostic tools, and the robustness of the diagnostic tools to severe errors. The maintainability scores were assigned by hand, and only took on three values: "low", "medium", and "high".

Although it shares the common technique of fault injection, our approach is quite different from EMC's. First, the EMC study focused more on the diagnosis issues of maintainability, examining the effectiveness of existing diagnostic procedures and tools, whereas our methodology is based on evaluation of known maintenance tasks. Both of these issues are important, however, and it would be interesting future work to merge the diagnosis focus into our methodology. Next, a more significant difference between the approaches is in how maintainability "scores" are assigned. Our methodology attempts to find objective measures based on directly-observable quantities: numbers of errors, times, and so forth, whereas EMC's uses a qualitative, coarse-grained score that is assigned by the experimenter. While these qualitative scores may be easier to assign, they also are potentially biased by the experimenter, and are not easy to compare across different benchmark experiments, unless the same person assigns the scores in all cases. In contrast, our metrics are more direct measures of the system itself, and thus we expect them to be independent of the experimenter. Finally, by using human subjects in our experiments, our methodology is able to capture more of the subtle interactions between system and user than is EMC's; for example, it would have been hard to arrive at most of our observations on the complexity of the administrative interfaces of Solaris, Linux, and Windows if we had only a documented procedure to follow, rather than human subjects actually trying to follow it.

Besides the EMC work, which like ours focused on direct experimental measurement of maintainability, most other approaches to examining maintainability have been based on surveys or models. Anderson [1], Dijker [15], and Kolstad [28] have all carried out surveys of how system administrators tend to spend their time; while these surveys help identify general areas where maintainability is lacking, they cannot provide direct insight into the maintainability factors of a particular system or set of systems. Kubicki's work on modeling the maturity of administration and maintenance processes suffers from similar problems [30]; it is useful more for understanding the general processes used by a site/installation than the maintainability of a particular system or set of systems, although it would be interesting future work to use some of Kubicki's ideas to augment our proposed use of task frequencies as a means of characterizing the maintainability workload of a given site or installation.

Turning now to the approach underlying our maintainability benchmark methodology, we find that the notion of application-specific benchmarking using orthogonal characterization axes is not a new one. In the simplest form, the notion can be seen in the processor-architecture community's CPI-based approach to evaluating processors, in which applications and processors are characterized by first building independent characterization vectors that measure the application usage and processor performance of each instruction in the processor's ISA, then combining those characterizations via a weighted average to get an overall performance score [24].

System characterizations in the form of vectors of cost metrics have been applied to more complex performance spaces as well. Saveedra and Smith used characterization vectors to characterize Fortran programs and Fortran runtime environments in terms of their primitive operations, allowing the performance of an arbitrary Fortran program running on an arbitrary Fortran runtime to be predicted with good success [40] [41]. Brown took a similar approach for benchmarking the performance of operating-systems-intensive applications, building independent characterization vectors for applications and operating systems, then combining them using a dot-product approach to get accurate performance predictions even when the application was not characterized on the same OS being used in the prediction [8]. Finally, Zhang, Seltzer, et al. have used characterization vectors as the foundation of an application-specific benchmarking framework, and have applied that framework to characterize the performance axes of the Java virtual machine and its garbage collector [42] [51] [52]. All of this characterization vector work has demonstrated the power of the application-specific benchmarking approach for independently quantifying various axes of the performance space, then merging them together via a cost function to get an overall, system- and application-dependent performance score; our maintainability benchmarking methodology is a natural extension of the approach to the maintainability space.

Section 5

Conclusions

In this report we have laid out frameworks for new kinds of benchmarks in two important areas left relatively unexplored by computer science researchers: availability and maintainability. We have defined metrics and general benchmarking methodologies for both availability and maintainability. We have demonstrated the efficacy of our general methodologies by specializing them to the study of software RAID systems, and then using them to unearth insights into the availability and maintainability properties of the Linux, Solaris, and Windows 2000 software RAID implementations.

Of the benchmarks we have described in this report, our availability benchmarks are probably the most accessible. They rely on a relatively simple methodology, produce easily-understood results, and require little extra infrastructure over existing performance benchmarks. And yet, despite their simplicity, they are quite powerful. As we saw in our case study of software RAID, our availability benchmark methodology was able to uncover each tested system's (undocumented) policy for mapping transient faults into failure conditions, and to quantify the impact of these policies and of the systems's failure recovery policies on the quality of service and availability delivered by I/O-intensive applications running on those systems. These are all insights that would be virtually unobtainable by traditional means, yet that are critically important to understand when deploying a high-availability service or system.

In contrast to the simplicity and directness of the availability benchmarks, our proposed maintainability benchmarking methodology suffers additional complexity due to its reliance on human experiments and an associated set of less-concrete metrics. Despite these complications, however, maintainability benchmarking is by no means a lost cause. Our case study shows its promise: even though we focused on one small maintenance task, and only looked at a subset of the metrics, we were able to draw qualitative and quantitative conclusions about the tested systems' maintainability along several axes. Most importantly, our benchmarks allowed us to identify system design factors that had a significant maintainability impact, including the structure and design of the system's maintenance interface, its resource naming convention, and its diagnostic reporting tools. Our results also lead us to more radical suggestions for improving system maintainability, by providing undo capabilities, built-in maintainability testing, and self-diagnosis and repair. By providing a way for system designers to quantitatively evaluate the maintainability impact of these (and other) design factors, even when applied in-the-small, our benchmarking techniques offer a fundamental tool for addressing what some see as today's maintainability crisis.

While we believe that the power of our benchmarking approaches is clearly illustrated in the insights that they uncovered, the research described in this report is only a first tentative step down what surely must be a long road to the important goal of comprehensive, portable, and meaningful benchmarks for availability, maintainability, and evolutionary growth. Whereas availability benchmarks surely have a head start, the immature state of both availability and maintainability benchmarking today is reminiscent of the early stages of performance benchmarking, although the challenges to be solved are quite different. We hope that growing interest into the challenges of availability and maintainability will bring more research and innovation, eventually leading to well-defined, practically-feasible benchmarks that fuse the demonstrated power of our techniques with the simplicity of today's mature performance benchmarks. We feel that reaching that goal is crucially important for the field, and we look forward to companionship on this journey.

Appendix A

Statistical Analysis of Maintainability TIME Data

In this appendix we describe the details of the paired-t and power analyses that we used in Section 3.4.1 to statistically analyze the TIME data from our maintainability experiments.

A.1 Paired-t Analysis

Recall that, in Section 3.4.1, our goal was to test the hypothesis that Solaris has a statistically better (lower) TIME score than Linux, and we intended to carry out this test by trying to reject the inverse null hypothesis H_0: $t_S \geq t_L$, since if we reject H_0, then we know that in fact $t_S < t_L$, which is what we want to show.

To examine H_0, we carry out a paired-t analysis, in which we return to the per-subject minimum times, and examine a data set consisting of the differences between minimum times, per subject, across the two OS's. That is, we take subject k's minimum time for Solaris and subtract his or her minimum time for Linux; the resulting value becomes the k'th value in our analysis data set. We work with the paired differences rather than the means or the unpaired minimums to work around the fact that each operating system likely has a different variance in the distribution of minimum times across subjects, a fact that is supported by the large disparities in standard deviation observed in the data in Figure 8 of Section 3.4.1. In the paired-t analysis, we assume only that each subject's *differences* are drawn from the same (presumed normal) distribution, a more likely situation.

To perform the paired-*t* analysis, we first compute a *t*-statistic for the pair differences. The *t*-statistic is given by:

$$t = \frac{\hat{\mu}}{S/\sqrt{n}}$$

where μ is the mean of the paired differences, S is the standard deviation of the paired differences, and *n* is the number of difference data points. For the Linux/Solaris case, $\mu = -16.75$, $S = 19.602$, and $n = 4$, resulting in a *t*-statistic of -1.709. We can now use the *t*-statistic to carry out a one-sided *t*-test of the hypothesis. The idea behind the test is that, if the hypothesis H_0 is true, and the assumptions are met, then the *t*-statistic will have a *t*-distribution on *n–1* degrees of freedom. Thus, under these assumptions, the probability that the *t*-statistic actually takes on the value that we computed above can be obtained by evaluating the probability distribution function of the *t*-distribution with $n-1$ degrees of freedom at the value of the *t*-statistic. The result is a quantity called a *P*-value.

In the case of Linux and Solaris, with the null hypothesis H_0: $t_S \geq t_L$, the *P*-value is given by $t_3(-1.709) \approx 0.093$. This indicates that there is a 9.3% chance that a *t*-statistic distributed according to the t_3 distribution would take on the value -1.709. In other words, it is reasonably unlikely that our *t*-statistic is in fact distributed according to the t_3 distribution, and thus it is reasonably likely that our null hypothesis H_0 is false, and that indeed $t_S < t_L$, as we were trying to show. Our confidence level in drawing this conclusion is only 90.7%, however, which is less than the typically-used standard of 95%.

Note that we carried out similar paired-*t* analyses for the comparisons of the other pairs of operating systems; the details are nearly identical to the analysis above, and the results are summarized in Figure 9 in Section 3.4.1.

A.2 Power Analysis

Besides the paired-*t* analysis, the other statistical analysis that was introduced in Section 3.4.1 is a *power analysis*, used to compute how many subjects would be needed in an experiment in order to obtain a result that could be supported at the 95% confidence level.

A power analysis is directly coupled with a statistical test, so in this case we consider a power analysis based on the paired-*t* hypothesis tests we described in the previous section.

The basic idea of a power analysis is to compute the sample size (number of subjects) needed to get the *P*-value of a statistical test below 0.05, the cutoff point for 95% confidence. If we assume that the mean μ and sample standard deviation *S* of the measured paired difference values are representative of what we would get in a larger experiment, then computing the sample size reduces to solving the following *P*-value equation for *n*:

$$P \; = \; t_{n-1}\left(\frac{\hat{\mu}\sqrt{n}}{S}\right) < 0.05$$

If we do this for the comparison of Linux and Solaris, we find that we would need $n \geq 6$ in order to get a *P*-value of less than 0.05, needed to reject the null hypothesis $H_0: t_S \geq t_L$ at the 95% confidence level. Similar analysis for a comparison between Linux and Windows suggests that we would need approximately 14 subjects to reject the null hypothesis $H_0: t_L \geq t_W$ with a *P*-value of less than 0.05. Finally, the power analysis for a comparison between Solaris and Windows suggests that 7 subjects would be sufficient to reject the null hypothesis $H_0: t_S \geq t_W$ with a *P*-value of less than 0.05.

Again, the results of our power analyses are summarized in Figure 9 in Section 3.4.1.

Appendix B

Qualitative Maintainability Observations

In this appendix, we describe our qualitative observations taken during the software RAID maintainability experiments described in Section 3.3. These notes are based on first-hand observations of the subjects as they performed the experiments, on analysis of the video-taped records of the experiments, and on comments made by the subjects, either unprompted or in post-experiment discussions.

B.1 Subject 1

At the start of the experiments, Subject 1 was given the training materials, but the subject spent very little time with them (less than 15 minutes). The first system that Subject 1 used was the Solaris system. The subject immediately began automating the repair procedure based on the training instructions, before a fault even occurred; the automation consisted of simple monitoring loops that used shell-based "while" commands to repeatedly invoke the RAID status-monitoring tool (metastat) and to watch the system log. Unfortunately, before any data could be collected, the Solaris system crashed due to problems with the experimental testbed, and we moved on to Windows.

Subject 1 had never used Windows 2000's administrative interface, but did not appear afraid of exploring the various menus and configuration options during the initial familiarization period and during the dead time waiting for faults. The subject successfully multitasked between reading e-mail and monitoring for faults. The only point where the subject seemed confused was when a disk that was originally reported as 47MB before a

failure came back as 49MB. We do not know why this behavior occurs, but it is repeatable.

Linux was the third system that Subject 1 used during the experiments. The subject seemed much more at home with this system than with Windows 2000 due to its command-line interface. The subject did not take any time for the familiarization period, and spent less than five minutes with the training slides. Again, the subject immediately began automating the tasks by writing scripts, and continued to work on the scripts during the idle periods in the tests (adding error checks, better argument processing, and so on). The subject wrote two scripts, one to handle the shutdown-restart cycle needed after the disk is replaced, and another that periodically extracted the RAID system status, parsed its output, and printed a message when a failure was detected. The subject seemed comfortable with the complex mapping from physical disks to logical disk names, and was careful to choose the correct disk to replace on a failure.

After the experiments had finished, we discussed the subject's impressions of the two systems. The biggest overall complaint was the lack of a good asynchronous failure notification mechanism on the systems; the subject felt that the current approach of having to poll the system for failure was inefficient. We talked about the differences between graphical interfaces (such as used by Windows 2000) and command-line interfaces (such as used by Linux). The subject was much more in favor of the command-line interface, since it could be scripted; the subject said that being able to write shell scripts is a big advantage, since in doing so the script writer is forced to think through the process and to anticipate how it might go wrong. The subject also felt that he better understood what was going on in the command-line case; Windows bothered this subject because "I [the subject] didn't know exactly what I was doing" when the subject clicked on various things. This is an interesting contrast with the second subject's experience, who had a similar reaction to the Linux system.

B.2 Subject 2

Subject 2 began the experiments with the Linux system. This subject did not write any scripts, but rather cleverly used four open xterms plus extensive use of the shell history and command substitution to make it easy to step through the repair task. This subject

relied heavily on pen and paper throughout the experiments, for example writing down the name of the disk that had failed and frequently referring back to it. The subject made several mistakes during the Linux tests, and twice removed the wrong disks. According to comments made during and after the tests, the cause of these mistakes was the baroque naming scheme that Linux uses for disks: when disks are being partitioned or added to the RAID volume, they are referred to as "/dev/sd[abcdef]"; these names map to the physical LUN identifiers 0–4 used in the subject's control GUI. But in the printout of RAID status and configuration, the disks are also numbered 0–4, but with a different mapping to the drive letters. The second incorrect disk removal was precisely due to the subject removing the disk based on the RAID status numbering rather than the physical numbering; the first was due to the subject apparently becoming flustered by getting error messages from accidentally running one of the repair steps out of order.

The subject's comments on Linux were that, at least in the subject's experience, the user interface affects the kind of mistakes that one makes. Most of the mistakes made by this subject were not logic errors, but were due to trying to cut and paste commands (or use shell history commands) and forgetting to update the disk name from the last time the command was used.

Subject 2 worked with Windows 2000 second, and seemed much more comfortable than with Linux. The subject claimed to be willing to try the repair task without the training notes, and in fact seemed to rely less on the notes than during the Linux test. However, the subject also noted that "my experience with Linux has affected my care with Windows," and this extra care (especially with respect to double-checking the disk names before the repair) is reflected by the subject's slower Windows times. Subject 2 particularly liked the Windows graphical interface, as it provided guidance through the task and constrained the possible choices at each step by removing inappropriate choices from the menus.

Subject 2 used Solaris last, and took a similar approach as with Linux. The subject again set up multiple xterms so that each xterm was responsible for one command or step in the repair sequence. With this setup, the subject could just fix up the disk name, and go around the xterms in order and always get the right sequence of events. This subject preferred Solaris to Linux, most importantly because Solaris used physical-based disk names

throughout, which made it very easy to correlate names with actual disks, unlike with Linux where two indirection steps were necessary to resolve the name. The subject also thought that Solaris's prompts and questions were easier to deal with than Linux's—under Linux, repartitioning the disk "seemed like black magic—type this, type that. Here [with Solaris] I had a better sense of what I was doing" since Solaris had more user-friendly prompts. The subject described Linux's RAID status reporting as "Linux typed all this junk at me" whereas Solaris's output was more easy to read and understand. Finally, the subject appreciated that Solaris had a lot fewer steps in its repair procedure than Linux.

Subject 2 commented "I feel like I'm in a hog-tying contest" when describing the effort to repeatedly and quickly repair the failures.

B.3 Subject 3

Subject 3 began with Windows 2000, and found its maintenance process to be very clear and straightforward. Like Subject 2, Subject 3 appreciated the fact that at each step, Windows presented a very small number of choices and that the constraints of the graphical interface made it difficult to enter erroneous input. However, the subject strongly disliked the fact that the numbers assigned to the disk by Windows did not match the physical disk identifiers (Windows uses a 1-based numbering system, while the physical identifiers are 0-based). The subject commented that "I had to keep double-checking myself so that I could be sure I had the right disk. I think that's one of the first things [I would] change. It was too easy to replace the wrong disk, especially if you're in a rush."

Subject 3 then proceeded to use the two command-line-based systems, starting with Solaris and then moving on to Linux. The subject's approach was similar to that of Subject 2, and involved arranged the necessary commands by xterms so that each xterm was responsible for one command or one block of commands in a phase of the repair procedure. This allowed the subject to proceed around the xterms using the recent command history, and thereby keep the order of the process straight. This worked quite effectively for Solaris, except that the subject felt that "the commands were all so similar" so it was a bit hard to keep them straight. The subject's other comment on Solaris was that it "seems that this [repair process] could be automated really easily." However, the subject made no

attempt to write scripts to automate the process. With Solaris, the subject appeared to be comfortable with the system after only a few iterations.

In contrast, Subject 3 had a lot of trouble with Linux, which the subject described at the end as "the most miserable" of the systems to use. The subject made two fatal errors with Linux, both of which resulted from the subject forgetting a crucial step (replacing the failed drive). When asked why this happened, the subject responded that it was very easy to lose track of the process: Linux required many steps and many different commands, yet there was no inherent guidance on how to proceed from one to the next. The subject felt that the interface "dissociated" the user from the actual repair process. Furthermore, it was easy to get off the script since there was no guidance in the interface on what tasks were valid (in terms of ordering, valid inputs, and so forth). Like Subject 2, Subject 3 also complained about Linux's baroque method of identifying disks. Although the numbering scheme did not directly cause any errors for Subject 3, this is likely because, unlike the other subjects, this subject asked many questions about the naming scheme during the initial training period, and thus was well aware of the complexities of the mapping scheme before starting the experiments. Despite this awareness, the naming scheme still appeared to be a source of some confusion, and slowed down the subject's progress during the actual experiments. Finally, Subject 3 felt that Linux's visual displays were confusing. The subject found it hard to tell when a failure had happened, and particularly to identify the failed disk. In contrast, "in Solaris it was much easier to see when [a disk] was failed, or rebuilding."

Overall, Subject 3 thought that Solaris and Windows were similar in their degree of difficulty, despite the fact that one was graphically-based and the other was command-line based. Solaris's command-line interface was not a drawback to the system because it was well-structured, presented information clearly, and required few commands to move through the process, especially compared to Linux. On balance, the subject thought Windows's interface was clearer, but Solaris's disk identification scheme was clearer.

B.4 Subject 4

Subject 4 started with the command-line systems, performing the experiments in the order Solaris, Linux, then Windows. For both of the command-line systems (Solaris and

Linux), the subject set up a simple shell-based while loop to repeatedly poll the system status, then read books during the dead time while waiting for a fault to occur. All of the commands needed to carry out the repair were performed in a single xterm window, without further scripting. The subject used shell history sporadically to speed up the process of reentering commands, but frequently simply re-typed the commands. When asked later as to why this approach was chosen, the subject commented that it was easier just to retype commands than to spend the time automating the tasks when the experiment was so short.

Subject 4 made two mistakes with Solaris. The first was due to disk naming issues: the subject was confused by the use of different partition identifiers for the array's data and metadata information. In this case, the subject recovered without experimenter intervention. The more serious mistake made by Subject 4 resulted when the subject accidentally skipped a step without realizing it, and continued with the later steps of the repair procedure. This put the system in a confusing state well off the path laid out in the instructions, and required experimenter intervention to recover. The subject blamed the command-line interface for the error, since it provided no feedback on the missed step and allowed the subject to proceed past it. In contrast, the subject felt that the graphical interface used by Windows would not have allowed this to happen.

Subject 4 had a similar problem with Linux during the last trial. The subject missed the fact that a command had reported an error, and continued executing further commands. Again, the command-line interface allowed this to happen, but in this case eventually flagged an error and the subject was able to detect the earlier error. Also, in the first iteration of the experiment, the subject made a potentially fatal error in specifying the disk name, but for some reason we have not been able to figure out, the system happened to accept it uncomplainingly. Like most of the other subjects, subject 4 bemoaned the complex naming scheme that Linux uses for its disks.

Subject 4 used the Windows system last, and immediately made a fatal error during the first iteration. The error in this case was again due to the common problem of disk naming convention: the subject did not realize that the on-screen disk numbering did not match the physical disk numbering, and therefore removed the wrong disk. Although Subject 4 was the only one to encounter this problem during our experiments, the sub-

ject's experience echoes Subject 3's comments about Windows's naming scheme. Finally, it was interesting to note that, while Subject 4 read printed reading materials while working with Solaris and Linux, the subject surfed the web during the Windows experiment. The subject's explanation was that Windows pops up a dialog box when a fault occurs, which is more likely to be noticed on-screen than the changed output of the text-based status tools used by Solaris and Linux. This observation reinforces Subject 1's point about the importance of asynchronous notification.

Subject 4's overall comments on the test were that disk numbering/identification systems were a problem (in Linux, with the disk letters, LUN numbers, and RAID slot numbers; in Windows, with the logically-assigned numbers; and in Solaris, with the multiple partitions), and that the subject liked Windows the best because of its graphical interface. The subject qualified his comments with the observation that the subject was performing the tasks manually, and not trying to automate them. The subject felt that Windows seemed less scriptable, but that for manual tasks it was easier to use because it did not require memorizing arcane commands, and because it restricted the choices at each step. The subject felt that "the GUI just sort of made sense" while the output of the other systems was more cryptic.

B.5 Subject 5

Subject 5 started the experiments with Linux. This subject appeared to be very comfortable with the command-line systems. With Linux, Subject 5 immediately began automating the fault detection process by writing a script that would poll for a failure and provide asynchronous failure notification by popping up an xterm window in the foreground. The subject wrote additional scripts to automate the entire repair process. Writing these scripts consumed a good deal of time during the first few iterations, but once they were deployed and working, the subject had very little work to do during the remaining iterations, resulting in low times and little opportunity for error. This subject made no errors at all during the Linux experiments.

Turning next to Solaris, Subject 5 again scripted the repair procedure extensively, more so even than Linux. The subject commented that the experience scripting Linux was probably useful in getting the Solaris automation working, although the subject did run

into a few difficulties in the early iterations due to bugs in the scripts. But by the third iteration of the Solaris experiments, the scripts were working and the subject needed to only interact three times with the system for each repair: an xterm would (asynchronously) pop up after a failure was detected and would poll until reconstruction completed. The subject then interacted once with the system, feeding a script the identity of the failed disk. The scripts then performed some work, notified the subject when it was safe to replace the disk, and, after the subject performed the disk replacement (the second interaction), the script carried out the remaining repair steps, pausing once for user input. After setting up these scripts, Subject 5 was able to surf the web, read mail, and work on other coding projects with no significant impact on his performance on the Solaris repair tasks.

Subject 5 was displeased with the final system, Windows 2000, because it required a significant amount of manual tending. The subject was not able to multitask during the reconstruction periods since there was no way to script the system to provide asynchronous notification of reconstruction completion. However, the subject did surf the web during the dead time between failures.

Although Subject 5 did not make any mistakes due to the disk naming conventions, the subject did comment that Solaris's was much easier to use and to script: "It's much easier to have the numbers in the disk name. Much less confusing." Solaris's naming scheme allowed for greater automation, since the disk number could be inserted directly into the appropriate commands. In contrast, Linux required a manual indirection to map disk letter to disk number.

Appendix C

Training Materials

This appendix reproduces the training materials that were given to our subjects during the software RAID maintainability experiments described in Section 3.3. The training materials are divided into four sections: a general overview of the RAID repair task and experimental environment, followed by three system-specific training packets.

C.1 General Training

Training for Software RAID Maintainability Experiments

Version 1.1
13 April 2000

Slide 1

Outline

- General overview of experiments
- Description of tasks to be performed
- Implementation-specific task details
  - Linux
  - Windows 2000
  - Solaris 7

Slide 2

Overview

- You will be maintaining pre-configured software RAID systems on 3 OS's:
  - Linux, Windows 2000, and Solaris
- Each system runs a web server whose data is stored on the RAID volume
- There will be several disk failures during the course of the experiments
- Your task: *detect and repair disk failures as they occur*

Slide 3

Refresher: Software RAID

- A software RAID system takes several individual disks and ties them together into a single logical volume
- We are using RAID-5, which:
  - uses redundant data storage to tolerate a single disk failure
  - requires a *rebuild* of data onto a fresh disk that is replacing a failed disk
  - often can automatically rebuild data onto a "hot spare" disk upon a failure

Slide 4

87

Your Tasks

1. **Detect failure**
   – all three systems require that you poll the system for status
2. **Restore redundancy by rebuilding data onto the spare disk**
   – may be manually- or automatically-initiated
3. **Replace failed disk with a new one**
4. **Activate new disk**
   – new disk either becomes a new spare, or the system rebuilds onto it and releases the old spare
   – activating a new disk may require stopping the web server and resetting the RAID software

Slide 5

Experimental Process

Familiarization Period

Issue Start command

Delay of up to 5 minutes

Disk fails

Linux, Solaris / Windows

System starts rebuild — User notices failure

User notices failure — User initiates rebuild

User waits for rebuild to complete

User removes old disk, inserts new disk

Issue Done Repair command — User activates new disk

Slide 6
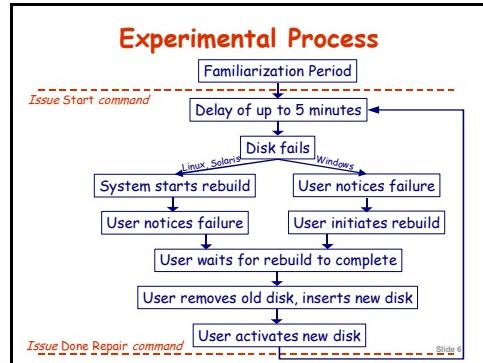
Experimental Setup

• **5-disk RAID-5 volume**
   – 4 disks hold active data
   – 5th disk starts out as a spare (hot or cold)

• **Disks are simulated**
   – all maintenance tasks (replacing disks, ...) are controlled by a control GUI

• **The control GUI is used to activate the experiment, "replace" simulated disks, and to mark your progress**

Slide 7

Experiment Control GUI

Control menu
(use to initialize
and quit)

Status information

Press to
start expts.

Press when
repairs are
complete

Use these buttons to simulate removing
a broken disk and inserting a new one

Slide 8
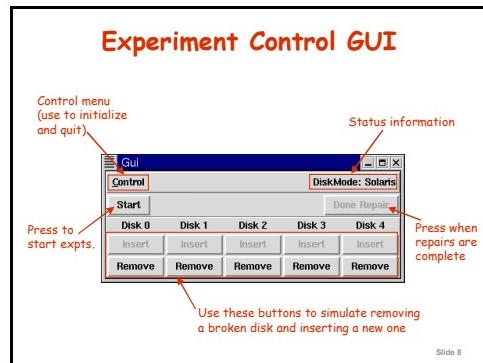
C.2 Linux-specific Training

Linux RAID Overview

• **Linux RAID managed by command-line tools**
   – your interface: an Xwindows session w/several xterms
• **The RAID system operates on *partitions***
   Disk 0: `/dev/sdb1`   (partition 1 on disk `sdb`)
   Disk 1: `/dev/sdc1`
   Disk 2: `/dev/sdd1`
   Disk 3: `/dev/sde1`
   Disk 4: `/dev/sdf1`   (initially a *hot* spare)
• **The logical RAID volume is** `/dev/md0`
   – mounted on `/raid`
• **The web server is apache**

Slide 10

Task 1: Detecting Failure

• **To poll for failure, run:** `cat /proc/mdstat`

• **Example normal output:**
```
Personalities : [raid0] [raid1] [raid5]
read_ahead 1024 sectors
md0 : active raid5 sdf1[4] sde1[3] sdd1[2] sdc1[1] sdb1[0] 153216
  blocks level 5, 32k chunk, algorithm 2 [4/4] [UUUU]
unused devices: <none>
```

• **Example failure output (disk 2, post-rebuild):**
```
Personalities : [raid0] [raid1] [raid5]
read_ahead 1024 sectors
md0 : active raid5 sdf1[2] sde1[3] sdd1[2](F) sdc1[1] sdb1[0] 153216
  blocks level 5, 32k chunk, algorithm 2 [4/4] [UUUU]
unused devices: <none>
```

Note: colors are for illustration only, and do not appear on-screen!

Slide 11

Task 2: Rebuilding Onto Spare

• **Rebuild is automatic in Linux**
• **Rebuild begins as soon as failure is detected**
• **During rebuild, running** `cat /proc/mdstat` **shows progress:**
```
Personalities : [raid0] [raid1] [raid5]
read_ahead 1024 sectors
md0 : active raid5 sdf1[4] sde1[3] sdd1[2](F) sdc1[1] sdb1[0] 153216
  blocks level 5, 32k chunk, algorithm 2 [3/4] [UU_U] recovery=45%
  finish=1.8min
unused devices: <none>
```
• **Detect end of rebuild by monitoring the output of** `cat /proc/mdstat:`
```
md0 : active raid5 sdf1[2] sde1[3] sdd1[2](F) sdc1[1] sdb1[0] 153216
  blocks level 5, 32k chunk, algorithm 2 [4/4] [UUUU]
```

Slide 12

Task 3: Replacing Disk

• *Do not start this step until rebuild completes!*
• **First, use control GUI to replace disk**
• **Then, partition and prep the new disk:**
   – example sequence for partitioning disk 3 (`/dev/sde`), user input in red:
```
# fdisk /dev/sde ↵
Command (m for help): n ↵
Command action
  e   extended
  p   primary partition (1-4)
p ↵
Partition number (1-4): 1 ↵
First cylinder (1-50, default 1): ↵
Last cylinder ... (1-50, default 50): ↵
Command (m for help): w ↵
```
   » error messages at the end of this sequence are normal

Slide 13

```
        Task 4: Activating New Disk
· First, shutdown the web server
   /usr/local/apache/bin/apachectl stop
· Then, reset the RAID software
   umount /raid
   raidstop /dev/md0
   raidstart /dev/md0
· Next, restart the web server
   mount /raid
   /usr/local/apache/bin/apachectl start
· Finally, add the new disk (disk 3 in this case)
   raidhotadd /dev/md0 /dev/sde1
      » the new disk becomes the new spare
                                        Slide 14
```

C.3 Windows 2000-specific Training

```
           Windows RAID Overview
· Windows RAID managed by graphical tools
   – access the interface via Start Menu/Programs/
     Administrative Tools/Computer Management/Storage/
     Disk Management (local)
· The RAID system operates on disks
   – disks are numbered sequentially on startup
· Windows RAID uses cold spares
   – rebuilds must be manually initiated
· The logical RAID volume is named RAID
   – mounted on R:\
· The web server is IIS
                                        Slide 16
```

```
           Task 1: Detecting Failure
· To poll for failure, watch the Windows screen
  for an error dialog box
   – failure is also reported by the state of the disks in
     the interface changing from "Online" to
     "Online (Errors)"
   – the state of the RAID volume R: will also change
     from "Healthy" to "Failed Redundancy"
                                        Slide 17
```

```
        Task 2: Rebuilding Onto Spare
· Rebuild must be manually initiated when
  failure is detected
   1. right-click on the disk icon labeled "RAID (R:)" in the
      upper panel of the disk management interface
   2. choose "Repair Volume..."
   3. select the cold spare disk and choose OK
· Screen shows progress of rebuild in lower
  panel
· Rebuild is done when status of RAID volume
  R: becomes "Healthy"
                                        Slide 18
```

```
           Task 3: Replacing Disk
· Do not start this step until rebuild completes!
· First, use control GUI to replace disk
· Then, scan for new disks
   – choose "Rescan Disks" from the "Action" menu at the
     top of the Windows screen
· The new disk appears at the same place as
  the old disk
   – the old disk is listed as "Missing" at the end of the
     list
       Hint: the "Properties" option obtained by right-clicking
       a disk icon on the left side of the lower panel will tell
       you which physical disk is represented by that icon
         (look for the LUN number in the box that appears)
                                        Slide 19
```

```
        Task 4: Activating New Disk
· Label the new disk
   – right-click on the box on the left side of the lower
     panel that shows the icon for the new disk
       » it will be labeled as "Unknown" and will have a white
         minus sign in a red circle
   – choose "Write Signature", select the disk, then "OK"
   – right-click the same icon, and choose "Upgrade to
     Dynamic Disk"
       » make sure the correct drive is selected before
         pressing OK
· The new disk is now the cold spare for the
  RAID volume
                                        Slide 20
```

C.4 Solaris-specific Training

Solaris RAID Overview

- **Solaris RAID managed by command-line tools**
  - *your interface: an Xwindows session w/several xterms*
- **The RAID system operates on** *partitions*
  - Disk 0: `c3t5d0s5` (partition `s5` on disk `c3t5d0`)
  - Disk 1: `c3t5d1s5`
  - Disk 2: `c3t5d2s5`
  - Disk 3: `c3t5d3s5`
  - Disk 4: `c3t5d4s5` (initially a *hot* spare)
  - *ancillary state kept on partition* `s3` *on each disk*
- **The logical RAID volume is** `/dev/md/dsk/d0`
  - *mounted on* `/raid`
- **The web server is apache**

Slide 22

Task 1: Detecting Failure

- **To poll for failure, run:** `metastat`
- **Example normal output:**

```
d0: RAID
  State: Okay
  [...]
  Device      Start Block  Dbase  State    Hot Spare
  c3t5d0s5         330       No    Okay
  c3t5d1s5         330       No    Okay
  c3t5d2s5         330       No    Okay
  c3t5d3s5         330       No    Okay
hsp001: 1 hot spare
  c3t5d4s5              Available       98304 blocks
```

- **After a failure, the hot spare changes state to "In Use" and shows up next to failed disk:**

```
  Device      Start Block  Dbase  State    Hot Spare
  c3t5d0s5         330       No    Okay
  c3t5d1s5         330       No    Okay     c3t5d4s5
  c3t5d2s5         330       No    Okay
  c3t5d3s5         330       No    Okay
hsp001: 1 hot spare
  c3t5d4s5              In use          98304 blocks
```

Slide 23

Task 2: Rebuilding Onto Spare

- **Rebuild is automatic in Solaris**
- **Rebuild begins as soon as failure is detected**
- **During rebuild,** `metastat` **shows progress:**

```
d0: RAID
  State: Resyncing
  Resync in progress: 54% done
  [...]
  Device      Start Block  Dbase  State      Hot Spare
  c3t5d0s5         330       No    Okay
  c3t5d1s5         330       No    Resyncing  c3t5d4s5
  c3t5d2s5         330       No    Okay
  c3t5d3s5         330       No    Okay

hsp001: 1 hot spare
  c3t5d4s5              In use          98304 blocks
```

- **Detect end of rebuild by monitoring** `metastat` **until "Resyncing" changes to "Okay"**
  - *don't go on to Task 3 until rebuild completes!*

Slide 24

Task 3: Replacing Disk

- **First, deactivate ancillary data on failed disk**
  `metadb -d c3t5d?s3`
- **Next, use control GUI to replace disk**
  - *Do not run* `metastat` *while disk is removed!!!*
- **Then, partition and prep the new disk:**
  - *example sequence for partitioning disk 3 (c3t5d3),
    with user input in* red:

```
# format -x /format.dat -t ASCvscsi -p RaidDisk \
  -d c3t5d3 ↵
format> fdisk ↵
[...]
Type "y" to accept the default partition, otherwise
 type "n" to edit the partition table.
y ↵
Warning: Solaris fdisk partition changed - Please
 relabel the disk
format> label ↵
Ready to label disk, continue? y ↵
format> q ↵
```

Slide 25

Task 4: Activating New Disk

- **First, prep the new disk's ancillary data**
  `metadb -a c3t5d?s3`

- **Then, notify the RAID software of new disk (in Solaris, this initiates an additional rebuild)**
  `metareplace -e d0 c3t5d?s5`

- **Finally, poll with** `metastat` **to determine when this rebuild has completed on the new disk**
  - *note that the original spare disk has returned to being a spare*

Slide 26

References

[1]   E. Anderson. Results of the 1995 SANS Survey. *;login: The USENIX Association Newsletter* 20(5), October 1995.

[2]   E. Anderson and D. Patterson. A Retrospective on Twelve Years of LISA Proceedings. *Proceedings of the 13th Systems Administration Conference*, Seattle, Washington, November 1999.

[3]   Anonymous et al. A Measure of Transaction Processing Power. *Datamation* 31(7): 112–118, 1985.

[4]   J. Arlat, A. Costes, et al. Fault Injection and Dependability Evaluation of Fault-Tolerant Systems. *LAAS-CNRS Research Report 91260*, January 1992.

[5]   R. Arpaci-Dusseau. Performance Availability for Networks of Workstations. *Ph.D. Dissertation*, U. C. Berkeley. December, 1999.

[6]   S. Asami. Reducing the Cost of System Administration of a Disk Storage System Built from Commodity Components. *Report No. UCB/CSD-00-1100, Computer Science Division, University of California at Berkeley*, May 2000.

[7]   ASC, Inc. Advanced Storage Concepts VirtualSCSI library. http://www.advstor.com/vscsi.html.

[8]   A. Brown. A Decompositional Approach to Computer System Performance Evaluation. *Technical Report TR-03-97, Harvard University Center for Research in Computing Technology*, Cambridge, MA, April 1997.

[9]   A. Brown, D. Oppenheimer, et al. ISTORE: Introspective Storage for Data-Intensive Network Services. *Proceedings of the 7th Workshop on Hot Topics in Operating Systems (HotOS-VII)*, Rio Rico, Arizona, March 1999.

[10] A. Brown and D.A. Patterson. Towards Availability Benchmarks: A Case Study of Software RAID Systems. *Proceedings of the 2000 USENIX Annual Technical Conference*, San Diego, CA, June 2000.

[11] J. Carreira, D. Costa, and J. Silva. Fault injection spot-checks computer system dependability. *IEEE Spectrum* 36(8):50–55, August 1999.

[12] S. Chandra and P. Chen. How Fail-Stop are Faulty Programs? In *Proceedings of the 28th International Symposium on Fault-Tolerant Computing*, June 1998.

[13] P. Chen, E. Lee, et al. RAID: High-Performance, Reliable Secondary Storage. *ACM Computing Surveys* 26(2):145–185, June 1994.

[14] R. Chillarege and N. Bowen. Understanding Large System Failure—A Fault Injection Experiment. In *Proceedings of the 1989 Fault-Tolerant Computing Symposium (FTCS)*, 356–363, 1989.

[15] B. Dijker. A Day in the Life of Systems Administrators. http://ditl.labyrinth.com/. Results presented during the SAGE BOF at LISA 98.

[16] Forrester. http://www.forrester.com/research/cs/1995-ao/jan95csp.html.

[17] J. Forrester and B. Miller. An Empirical Study of the Robustness of Windows NT Applications Using Random Testing. In *Proceedings of the 4th USENIX Windows System Symposium*, Seattle, WA, August 2000.

[18] A. Fox, S. Gribble, et al. Cluster-Based Scalable Network Services. In *Proceedings of the 16th Symposium on Operating System Principles*, St. Malo, France, October 1997.

[19] Gartner. http://www.gartner.com/hcigdist.htm.

[20] J. Gray. Locally served network computers. Microsoft Research white paper. February 1995. Available from http://research.microsoft.com/~gray.

[21] S. Gribble, E. Brewer, et al. Scalable, Distributed Data Structures for Internet Service Construction. *Proceedings of the 4th Symposium on Operating Systems Design and Implementation (OSDI 2000)*, 319–332, San Diego, CA, October 2000.

[22] J. Hamilton. Remarks made at the feedback session of the 2000 Semi-annual UC Berkeley IRAM/ISTORE Research Retreat, July 2000.

[23] J. Hennessy. The Future of Systems Research. *IEEE Computer* 32(8):27–33, August 1999.

[24] J. Hennessy and D. A. Patterson. *Computer Architecture: A Quantitative Approach, 2nd Ed.* Boston: Morgan Kaufmann Publishers, 1996.

[25] Y. Huang, Z. Kalbarczyk, and R. Iyer. Dependability analysis of a cache-based RAID system via fast distributed simulation. In *Proc. 17th IEEE Symposium on Reliable Distributed Systems*, October 1998.

[26] M. Kaâniche, L. Romano, et al. A Hierarchichal Approach for Dependability Analysis of a Commercial Cache-Based RAID Storage Architecture. In *Proceedings of 28th International Symposium on Fault Tolerant Computing*, June 1998.

[27] W. Kao, R. Iyer, and D. Tang. FINE: A Fault-Injection and Monitoring Environment for Tracing the UNIX System Behavior under Faults. *IEEE Trans. Software Eng.*, 19(11):1105–1118, November 1993.

[28] R. Kolstad. 1992 LISA Time Expenditure Survey. *;login:.* Available at http://www.cs.pdx.edu/~trent/sysadmin/docs/time-summary.

[29] P. Koopman, J. Sung, et al. Comparing Operating Systems Using Robustness Benchmarks. In *Proceedings of the 16th Symposium on Reliable Distributed Systems*, 72–79, October 1997.

[30] C. Kubicki. The System Administration Maturity Model—SAMM. *Proceedings if the Seventh Systems Administration Conference (LISA '93)*, 213–225, Monterey, California, 1993.

[31] D. Lambright. Experiences in Measuring the Reliability of a Cache-Based Storage System. *Proceedings of the First Workshop on Industrial Experiences with Systems Software (WIESS 2000)*, 11–20, San Diego, CA, October 2000.

[32] B. Miller, L. Fredriksen, and B. So. An Empirical Study of the Reliability of UNIX Utilities. *Communications of the ACM*, 33(12):32–43, December 1990.

[33] B. Miller, D. Koski, et al. Fuzz Revisited: A Re-examination of the Reliability of UNIX Utilities and Services. *Technical Report CS-TR-95-1268, University of Wisconsin, Madison*, 1995.

[34] B. Murphy and T. Gent. Measuring System and Software Reliability using an Automated Data Collection Process. *Quality and Reliability Engineering International*, 11(5): 341–353, 1995.

[35] W. Ng and P. Chen. The Systematic Improvement of Fault Tolerance in the Rio File Cache. In *Proceedings of the 1999 Symposium on Fault-Tolerant Computing*.

[36] M. Richtel. Keeping E-Commerce On Line; As Internet Traffic Surges, So Do Technical Problems. *The New York Times*, 21 June 1999.

[37] SAGE System Administrator's Guild. *SAGE Certification.* http://www.usenix.org/sage/cert/.

[38] Y. Saito, B. Bershad, and H. Levy. Manageability, availability and performance in Porcupine: a highly scalable, cluster-based mail service. *Proceedings of the 17th Symposium on Operating System Principles*, Kiawah Island, South Carolina, December 1999.

[39] M. Satyanarayanan. *Digest of Proceedings, Seventh IEEE Workshop on Hot Topics in Operating Systems (HotOS-VII)*, Rio Rico, AZ, March 29–30, 1999.

[40] R. Saavedra-Barrera, A. Smith, and E. Miya. Machine Characterization Based on an Abstract High-Level Language Machine. *IEEE Transactions on Computers*, 38(12):1659–1679, December 1989.

[41] R. Saavedra and A. Smith. Analysis of Benchmark Characteristics and Benchmark Performance Prediction. *ACM Transactions on Computer Systems* 14(4):344–384, November 1996.

[42] M. Seltzer, D. Krinsky, et al. The Case for Application-Specific Benchmarking. *Proceedings of the 1999 Workshop on Hot Topics in Operating Systems (HotOS-VII)*, Rio Rico, AZ, March 1999.

[43] K. Shanley. History and Overview of the TPC. http://www.tpc.org/articles/tpc.overview.history.1.html.

[44] D. Siewiorek, J. Hudak, et al. Development of a Benchmark to Measure System Robustness. In *Proceedings of the 1993 International Symposium on Fault-Tolerant Computing*, 88–97, June 1993.

[45] SPEC, Inc. *SPECweb99 Benchmark.* http://www.spec.org/osg/web99.

[46] SPEC, Inc. *SPECcpu Benchmark.* http://www.spec.org/osg/cpu2000.

[47] N. Talagala. Characterizing Large Storage Systems: Error Behavior and Performance Benchmarks. *Ph.D. Dissertation*, U. C. Berkeley. September, 1999.

[48] R. Thomas, N. Talagala. What Happens Before a Disk Fails. Talk at the Winter 1999 IRAM Semi-annual Research Retreat. January, 1999.

[49] Transaction Processing Performance Council. http://www.tpc.org.

[50] T. Tsai, R. Iyer, and D. Jewett. An Approach towards Benchmarking of Fault-Tolerant Commercial Systems. In *Proceedings of the 1996 Symposium on Fault-Tolerant Computing (FTCS)*, June 1996.

[51] X. Zhang and M. Seltzer. HBench:Java: An Application-Specific Benchmarking Framework for Java Virtual Machines. *ACM Java Grande 2000*, San Francisco, California, June 2000.

[52]   X. Zhang and M. Seltzer: HBench:JGC: An Application-Specific Benchmark Suite for Evaluating JVM Garbage Collector Performance. *Proceedings of the 6th USENIX Conference on Object-Oriented Technologies and Systems (COOTS '01)*, to appear.